VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

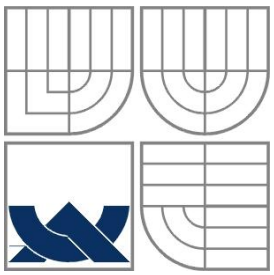
VYHLEDÁVÁNÍ NEJDELŠÍHO SHODNÉHO PREFIXU VE VYSOKORYCHLOSTNÍCH SÍTÍCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

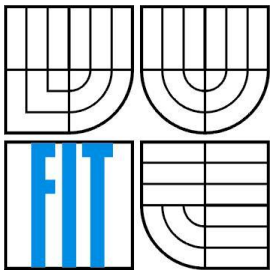
AUTOR PRÁCE
AUTHOR

Bc. MARTIN SKAČAN

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYHLEDÁVÁNÍ NEJDELŠÍHO SHODNÉHO PREFIXU VE VYSOKORYCHLOSTNÍCH SÍTÍCH

LONGEST PREFIX MATCH IN HIGH-SPEED NETWORKS

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE — Bc. MARTIN SKAČAN
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KOŘENEK, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se zabývá vyhledáváním nejdelšího shodného prefixu (LPM), což je časově kritická operace při směrování paketů. Pro dosažení propustnosti 100Gbps je nutná hardwarová implementace této operace a směrovací tabulka musí být uložena v paměti na čipu, která je omezena nízkou kapacitou. Současné LPM algoritmy vyžadují velké množství paměti pro uložení směrovacích tabulek protokolu IPv6, nebo je není možno jednoduše implementovat v HW. Proto jsem se zaměřil na analýzu směrovacích tabulek IPv6 a několika známých LPM algoritmů. Na základě této analýzy jsem navrhl nový algoritmus, který vyniká nízkou paměťovou složitostí pro IPv4/IPv6 vyhledávání. Navržený algoritmus má nejnižší paměťové nároky v porovnání s existujícími LPM algoritmy. Navíc je vhodný pro nasazení ve vysokorychlostních 100Gbps sítích, což bylo ukázáno s pomocí nové hardwarové architektury využívající zřetězené zpracování s propustností 140Gbps.

Abstract

This thesis deals with the Longest Prefix Matching (LPM), which is a time-critical operation in packet forwarding. To achieve 100Gbps throughput, this operation has to be implemented in hardware and a forwarding table has to fit into the on-chip memory, which is limited by its small size. Current LPM algorithms need large memory to store IPv6 forwarding tables or cannot be simply implemented in HW. Therefore we performed an analysis of available IPv6 forwarding tables and several LPM algorithms. Based on this analysis, we propose a new algorithm which is able to provide very low memory demands for IPv4/IPv6 lookups. To the best of our knowledge, the proposed algorithm has the lowest memory requirements in comparison to existing LPM algorithms. Moreover, the proposed algorithm is suitable for IP lookup in 100Gbps networks, which is shown on new pipelined hardware architecture with 140Gbps throughput.

Klíčová slova

LPM, nejdelší shodný prefix, IP sítě, IPv6, vyhledání IP, algoritmy, paměť, propustnost.

Keywords

LPM, longest prefix match, IP networks, IPv6, IP lookup, algorithms, memory, throughput.

Citace

Martin Skačan: Vyhledávání nejdelšího shodného prefixu ve vysokorychlostních sítích, diplomová práce, Brno, FIT VUT v Brně, 2013

Vyhledávání nejdelšího shodného prefixu ve vysokorychlostních sítích

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Kořenka, Ph.D. Další informace mi poskytli členové vývojové skupiny ANT@FIT. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Martin Skačan
16. května 2013

Poděkování

Chtěl bych poděkovat vedoucímu své diplomové práce Ing. Janu Kořenkovi, Ph.D. za kvalitní vedení, odbornou pomoc, připomínky a návrhy, kterými mě podporoval v práci. Dále bych rád poděkoval Ing. Jiřímu Matouškovi za odbornou spolupráci, rady a poskytnuté informace.

Obsah

1 Úvod

Internet, postavený na robustnom IP protokole, je dnes považovaný za najdostupnejší zdroj informácií. Táto obrovská sieť je súčasťou všetkých odvetví ľudskej činnosti a už dávno neponúka len jednoduchú komunikáciu v podobe e-mailov a webových stránok. Jej potenciál a možnosti ešte zďaleka nie sú vyčerpané a má čo ponúknuť i do budúcnosti.

Dostupnosť širokopásmového internetu vyvolala u užívateľov zvýšené požiadavky na multimediálne služby, akými sú sledovanie televíznych programov vo vysokom rozlíšení (CATV), sťahovanie alebo streamovanie hudby a videa, online hry, vzdialená podpora a učenie, VoIP, videokonferencie a mnohé ďalšie. Internetový tok dát dosahuje extrémne hodnoty a naďalej rapídne stúpa. Ďalším faktorom, s ktorým sa internet musí vysporiadať, je enormný nárast užívateľov a koncových zariadení, ktoré nepredstavujú len počítače. Aktuálnym trendom je rozšírenie smartphonov, netbookov, tabletov a iných mobilných či vstavaných zariadení, ktoré disponujú internetovým pripojením. S týmto rozmachom musia neustále držať krok poskytovatelia internetu a dodávatelia sieťových prvkov. Pre poskytovanie kvalitných služieb je nevyhnutné zabezpečiť dostatočnú rýchlosť liniek a vysokú priepustnosť v sieťových zariadeniach.

Problém priepustnosti sa objavuje najčastejšie v smerovačoch. Tie musia každý prichádzajúci paket klasifikovať na základe IP adresy a podľa výsledku preposlať ďalej. Čas, ktorý je potrebný na prechod paketu zariadením, predstavuje spracovanie hlavičiek, vyhľadanie IP adresy v smerovacej tabuľke a preposlanie paketu na výstup. K tomu sa často pridávajú režijné operácie pre zaistenie bezpečnosti, meranie štatistík a ďalšie úkony, ktoré zvyšujú nároky na spracovanie paketov. Časovo kritickou operáciou, ktorá má zásadný vplyv na celkovú výkonnosť je však vyhľadanie IP adresy v smerovacej tabuľke. Práve k tomu sa využívajú algoritmy pre vyhľadávanie najdlhšieho zhodného prefixu - LPM. Od vhodne zvoleného a optimalizovaného algoritmu sa potom odvíja celá rada vlastností smerovača a do istej miery určuje jeho limity. Pre kontinuálny vývoj internetových sietí a posúvanie hraníc je potrebné venovať sa optimalizovaniu úlohy LPM.

Existuje množstvo LPM algoritmov, založených na rôznych princípoch a architektúrach. Prevažná väčšina je však optimalizovaná pre prácu s protokolom IPv4. Tento protokol už mnohokrát prekročil očakávanú dĺžku svojho života. Napriek tomu si stále drží svoje prvenstvo a nástup protokolu IP verzie 6 postupuje pozvoľne. Tento pomalý prechod je problémom skôr ekonomickým ako technickým. Aktuálny trh prejavuje nevôľu akceptácie nového protokolu, hlavne z finančného hľadiska. Z pohľadu spoločností predstavuje zvýšené náklady, za ktoré nedostávajú (dostatočnú) pridanú hodnotu, pretože súčasné riešenie stále funguje. Ak sa určitá spoločnosť rozhodne používať nový protokol, nie je možné jednoducho ignorovať predchádzajúcu verziu. Je potrebné zabezpečiť duálnu podporu oboch protokolov, smerovače a sieťové prvky musia podporovať nové funkcie a služby. To všetko a mnohé ďalšie faktory brzdili nasadenie protokolu IPv6. Napriek tomu je jeho masívne rozšírenie len otázkou času a tento prechod je nevyhnutný. Preto sa v mojej práci orientujem práve na protokol IPv6.

Príchod IPv6 predstavuje nové výzvy a problémy aj v oblasti vyhľadávania najdlhšieho zhodného prefixu. Staršie známe algoritmy, pri pokuse aplikovať ich na nový protokol, spravidla zlyhávajú a stávajú sa nepoužiteľné. Alternatívy pre IPv6 poskytujúce porovnateľný pomer cena/výkon sú značne obmedzené. Prioritou je pritom vysoká priepustnosť riešenia, ktorá bude dostatočná v reálnom nasadení. Pre dosiahnutie priepustnosti na úrovni 100Gbps je nutné vykonať až 160 miliónov vyhľadaní za sekundu. Dostávame teda hodnotu približne 6 ns vyhradených na

uskutočnenie jedného vyhľadania. Výkon počítačových čipov je dnes na dostatočnej úrovni a vďaka HW akcelerácii dokážeme dosiahnuť dokonca vyššie hodnoty. Úzkym hrdlo sa ale spravidla stáva pomalý prístup do pamäte, ktorý brzdí celkový výkon. Riešením by mohlo byť využívanie výhradne internej pamäte na čipe, ktorá má vysokú prenosovú rýchlosť, nízku spotrebu a predovšetkým nízku latenciu. Problémom však je, že takéto pamäte majú obmedzenú kapacitu. Súčasné LPM algoritmy zvyčajne vykazujú vysokú pamäťovú náročnosť, a preto k svojej práci vyžadujú použitie pomalej externej pamäte. Niektoré ďalšie typické problémy LPM algoritmov je nedostatočná rýchlosť pri protokole IPv6, nízka škálovateľnosť s počtom prefixov, prípadne náročná alebo neexistujúca HW realizácia. Navrhnúť riešenie, ktoré dokáže efektívne pracovať vo všetkých podmienkach, nie je jednoduché.

Z toho dôvodu som si za hlavný cieľ mojej práce stanovil návrh nového LPM algoritmu, ktorý bude efektívne pracovať s adresami IPv6 a zaručí dostatočnú priepustnosť. Prioritou bolo umožniť HW implementáciu a predovšetkým minimalizovať pamäťové nároky do takej miery, aby bolo možné uložiť všetky potrebné dátové štruktúry do internej pamäti na čipe. Uvedené ciele sa mi podarilo úspešne splniť vďaka novému spôsobu reprezentácie prefixovej množiny. Navrhol som nové typy používaných uzlov, ktoré sú optimalizované na základe vykonanej analýzy LPM problematiky. Navrhnuté riešenie prekonáva všetky známe algoritmy pri práci s protokolom IPv6 a samozrejme tiež pri protokole IPv4. Zároveň bola navrhnutá odpovedajúca hardwarová architektúra, ktorá disponuje priepustnosťou až 140Gbps.

Text tejto práce je rozdelený do viacerých logických celkov. Prvá časť predstavuje teoretický rozbor problematiky LPM, ktorá podáva všetky potrebné vedomosti a pomáha dobre sa zorientovať v danej oblasti. Ďalšia časť práce obsahuje vybrané typy algoritmov, ktoré sú bližšie prezentované. Sústredil som sa na vysvetlenie základných dátových štruktúr a podstatných myšlienok, na ktorých sú postavené. Záver kapitoly ukazuje stručné zhrnutie a porovnanie uvedených metód. Tieto úvodné kapitoly boli vypracované v rámci semestrálneho projektu, na ktorý táto diplomová práca nadväzuje. Štvrtá kapitola sa venuje podrobnej analýze súčasného stavu. Zameriavam sa jednak na analýzu uvedených algoritmov, ale hlavne na prefixové množiny s cieľom identifikovať ich charakteristické vlastnosti. Na základe tejto analýzy vznikal návrh nového algoritmu, ktorý je podrobne popísaný v nasledujúcej kapitole. Priblížený je návrh novej reprezentácie, spôsob práce a tiež navrhnutá HW architektúra. V kapitole 6 a 7 sú potom prezentované experimenty a výkonnostné zhodnotenie navrhnutého riešenia. Záverečná kapitola nakoniec hodnotí a diskutuje dosiahnuté výsledky a stručne popisuje možnosti ďalšieho vývoja.

2 Teoretický rozbor

Táto kapitola poskytuje stručný úvod do problematiky a obsahuje potrebné znalosti a prerekvizity nutné pre správne pochopenie nasledujúcich častí práce. Nesnaží sa však podávať vyčerpávajúci výklad. V prípade nejasností je možné na doplnenie vedomostí využiť citovanú odbornú literatúru.

Cieľom kapitoly je predovšetkým zasadiť problematiku LPM do širšieho kontextu. Zároveň by mala poskytnúť čitateľovi predstavu o tom, v ktorej oblasti informatiky sa táto úloha nachádza a zdôrazniť, pre ktoré oblasti je nevyhnutná. V jednotlivých podkapitolách prechádzame postupne z vyššieho pohľadu abstrakcie až ku konkrétnemu problému.

2.1 Vrstvový model TCP/IP

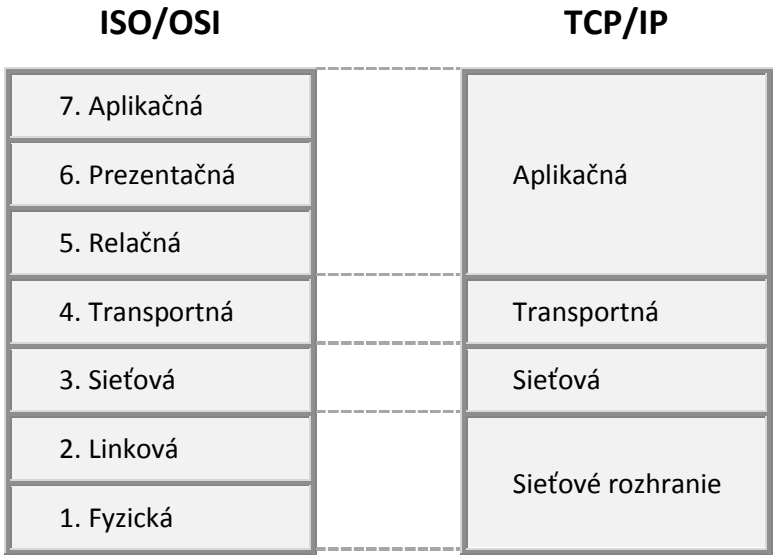
Činnosť počítačových sietí sa najlepšie vysvetľuje na vrstvovom modeli architektúry. V súčasnosti sa využívajú 2 najčastejšie modely a to je ISO/OSI model a model TCP/IP.

Model OSI (Open System Interconnection) vyvinula Medzinárodná štandardizačná organizácia (ISO – International Standards Organization) pre potreby otvoreného prepojovania počítačových systémov. Tento model je považovaný za referenčný a identifikuje všetky funkcie potrebné pre naviazanie, používanie, definovanie a zrušenie komunikačnej relácie medzi dvoma počítačmi. Tieto funkcie usporadúva do 7 logicky definovaných vrstiev. Model OSI je považovaný za primárnu architektúru pre počítačovú komunikáciu a vychádza z neho mnoho dnešných protokolov. V praxi sa však implementovala iba časť tohto modelu.

Model TCP/IP je založený na protokoloch TCP (Transmission Control Protocol) a IP (Internet Protocol). Tento model vznikol pod vedením agentúry DARPA (Defence Advanced Research Project Agency). Jej cieľom bolo vyvinúť spoľahlivú komunikačnú sieť, ktorá mala byť decentralizovaná, robustná, nezávislá na prenosovom médiu a jednoducho implementovateľná. Práve tento model je štandardom Internetu a je dnes implementovaný vo väčšine počítačových sietí.

Oba modely sú veľmi podobné a ich porovnanie zobrazuje obrázok 2.1. Vidíme, že model TCP/IP je výrazne jednoduchší, pretože spája služby prvé troch vrstiev do aplikačnej vrstvy. Podobne fyzickú a linkovú vrstvu modelu OSI spája do vrstvy fyzického rozhrania. Týmto odstraňuje nedostatky komplikovanej štruktúry modelu OSI.

Každá vrstva modelu popisuje pravidlá pre prenos dát medzi procesmi rovnakej vrstvy. Nie je definovaný jeden konkrétny protokol, ale sú určené funkcie dátovej komunikácie, ktoré musia protokoly na vrstve vykonávať. Pritom sa využívajú služby poskytované nižšími vrstvami. Počas komunikácie vidíme logický tok dát medzi procesmi na rovnakej vrstve. V skutočnosti ale reálny tok prebieha vždy vertikálne cez všetky vrstvy. Pri odosielaní dát každá vrstva pridá k údajom svoju hlavičku a predá výsledok nižšej vrstve. Nakoniec prebehne samotný fyzický prenos na najnižšej úrovni modelu a na strane prijímateľa sa zase vykonáva rozbaľovanie dát v opačnom poradí.



Obrázok 2.1: Porovnanie vrstiev modelov ISO/OSI a TCP/IP

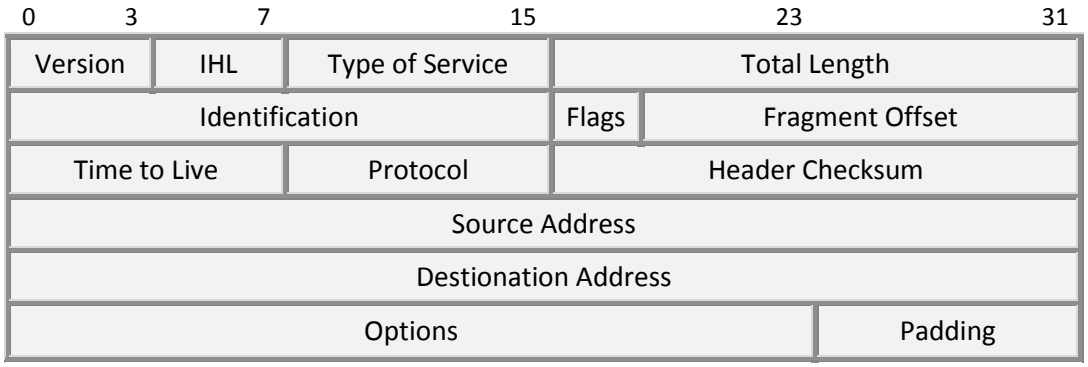
Podrobný popis všetkých vrstiev modelu TCP/IP alebo ISO/OSI nájdeme takmer v každej publikácii o počítačových sieťach [6]. Pre túto prácu je však najpodstatnejšia sieťová vrstva, nazývaná tiež internetová alebo IP vrstva. Na tejto vrstve prebieha zabaľovanie dát do tzv. datagramov, ktoré sa prenášajú na miesto určenia. Toto doručovanie zabezpečujú smerovacie protokoly, ktoré sú zodpovedné za vytvorenie cesty medzi zdrojovým a cieľovým počítačom. Dominantným protokolom sieťovej vrstvy sa stal IP protokol, ktorý sa dnes používa súbežne v 2 verziách.

2.2 IP protokol

Pôvodná verzia protokolu IP verzie 4 bola definovaná v RFC 791 [13]. Hlavnou úlohou tohto protokolu je zabezpečenie priechodu dátovej komunikácie cez rôzne siete a doručenie paketov až do cieľového zariadenia. Súčasťou protokolu však nie sú žiadne funkcie potvrdzovania, riadenia toku alebo zoraďovania paketov. Všetky tieto funkcie sú prenechané na protokoly vyšších vrstiev, napríklad TCP.

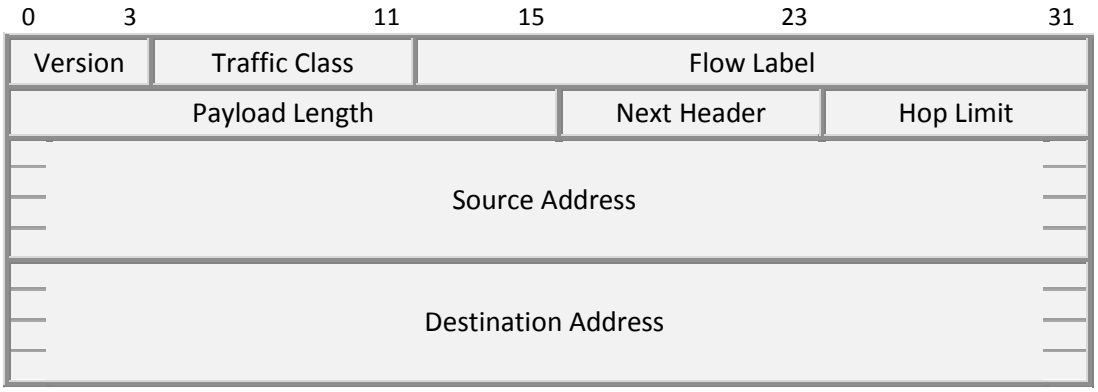
Jednotlivé siete, sieťové zariadenia a počítače zapojené v sieti identifikuje protokol IPv4 pomocou unikátnej 32 bitovej IP adresy. Táto hodnota spolu s množstvom ďalších údajov je súčasťou hlavičky každého paketu, ktorú vidíme na obrázku 2.2. IP adresa sa zapisuje ako štyri 8-bitové čísla v dekadickom formáte oddelené bodkou (napr. 192.168.16.4). Tento protokol, ktorý si drží aj po približne 30 rokov dominantné postavenie má však mnohé nedostatky a obmedzenia. Hlavným z nich je malý adresný priestor – teoreticky 2^{32} unikátnych adries. Niektoré bloky adries sú navyše rezervované pre špeciálne použitie. S vývojom Internetu a prudkým nárastom jeho užívateľov bolo nutné revidovať spôsoby prideľovania adries, aby nedochádzalo k ich plytvaniu. Napriek mnohým snahám (napr. použitie CIDR alebo NAT) však došlo k vyčerpaniu priestoru IPv4 adries[1].

[1] Dňa 3.2.2011 organizácia IANA pridelila posledné voľne /8 bloky regionálnym organizátorom, ktorí ich môžu rozdeliť na podbloky a prideliť žiadateľom vo svojom regióne.



Obrázok 2.2: Ukážka hlavičky protokolu IPv4

Protokol IPv6 je stavaný ako jednoduchá, dopredu kompatibilná, náhrada protokolu IPv4. Snaží sa odstrániť nedostatky predchádzajúcej verzie a tiež poskytnúť nové možnosti a služby pre dnešné moderné siete. Pri pohľade na hlavičku nového protokolu (obr. 2.3) vidíme mnohé rozdiely. Určite si všimneme, že hlavička je výrazne jednoduchšia, čím sa znižujú nároky na spracovanie paketu. Pritom však za svojím predchodcom funkčne nezaostáva, ale práve naopak. Poskytuje navyše nové možnosti a funkcie, ako napríklad zvýšenie bezpečnosti, podpora autokonfigurácie, efektívna hierarchia adries a smerovania, podpora QoS (Quality of Service) a ďalšie [10].



Obrázok 2.3: Ukážka hlavičky IPv6

Podstatným rozdielom je ale hlavne zvýšenie šírky zdrojovej a cieľovej IP adresy na 128 bitov. To nám ponúka nepredstaviteľne veľký adresný priestor 2^{128} (~ $3{,}4x10^{38}$) adries. Z toho je vyše 85% rezervovaných pre zatiaľ nešpecifikované využitie v budúcnosti [7]. Zmenil sa tiež zápis nových adries – adresa je rozdelená na 8 častí, ktoré sú oddelené dvojbodkou. Každá časť teda predstavuje 16 bitov adresy a je navyše zapísaná v hexadecimálnom formáte (napr. 1080:0:0:0:0:0:200C:417A). Z dôvodu použitých alokačných metód je pre adresy IPv6 typické, že obsahujú dlhé reťazce nulových bitov. Túto neprerušenú postupnosť núl v adrese môžeme vynechať a využiť skrátený zápis (1080::200C:417A). Toto skrátenie (::) sa v každej adrese môže vyskytovať maximálne jedenkrát. V opačnom prípade by sme neboli schopní určiť koľko bitov bolo vynechaných v jednotlivých častiach.

2.3 Smerovanie v IP sieťach

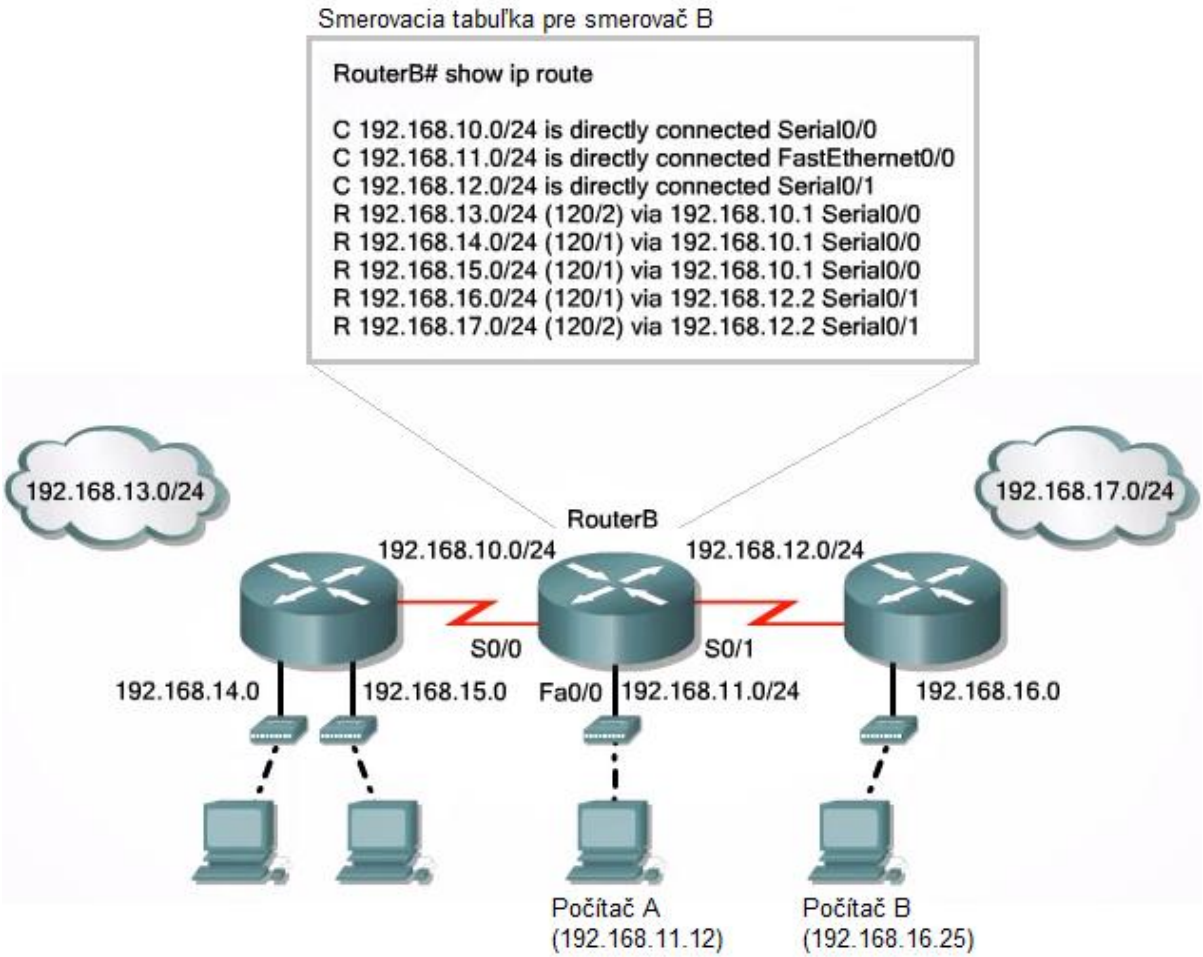
Jednou zo základných operácií, ktoré vykonáva sieťová vrstva je smerovanie dátových paketov. Je to veľmi komplikovaný proces, z ktorého si popíšeme len základné princípy. Bude vysvetlené, akým spôsobom jednotlivé sieťové zariadenia vyberajú správnu cestu pre naše dáta.

Komunikácia medzi počítačmi v rámci jednej lokálnej siete je pomerne jednoduchá a nevyžaduje príliš zložité techniky. Avšak smerovacie protokoly musia zabezpečiť doručenie paketov aj do počítačov vzdialených tisícky kilometrov, pričom dopredu nepoznáme cestu alebo počet sprostredkovateľských zariadení medzi nimi. Sieťové zariadenia, ktoré hrajú úlohu sprostredkovateľa, sú najčastejšie smerovače. Na rozdiel od niektorých iných prvkov (napr. most alebo prepínač) dokáže smerovač pracovať na sieťovej vrstve. Vďaka tomu môže vzájomne prepájať rôzne siete prostredníctvom adresovania v IP.

Každý smerovač musí mať viacero fyzických rozhraní, ktoré sú pripojené k jednotlivým sieťam. Behom svojej činnosti zisťuje smerovač adresy počítačov a sietí, ktoré sú k nemu pripojené a ukladá tieto informácie do tabuľky. Táto smerovacia tabuľka teda definuje vzťahy medzi IP adresami a portami fyzického rozhrania, na ktorých sú dané systémy pripojené. Ukážku smerovacej tabuľky a odpovedajúcej sieťovej topológie vidíme na obrázku 2.4. Tabuľku podobnej štruktúry obsahuje každé sieťové zariadenie a počítač pripojený k sieti. Pri bežnej činnosti smerovač prijme paket a spracuje jeho hlavičku, z ktorej si prečíta IP adresu cieľového počítača. Následne vo svojej smerovacej tabuľke vyhľadá najvhodnejší záznam a prepošle prijatý paket na daný port fyzického rozhrania. Týmto spôsobom pakety putujú medzi smerovačmi až sa dostanú do cieľovej siete a ku koncovému zariadeniu. V prípade, že adrese nevyhovuje žiadny záznam z tabuľky, je paket typicky preposlaný na adresu implicitnej brány. Ak táto nie je v smerovacej tabuľke zadaná, bude paket zahodený.

Uvažujme, že počítač A (192.168.11.12) na obrázku sa rozhodne poslať správu počítaču B (192.168.16.25). Počítač A vytvorí paket a podíva sa do svojej smerovacej tabuľky. Tam pravdepodobne nenájde vhodný záznam, pretože počítač B sa nachádza v inej podsieti. Preto odošle paket na svoju implicitnú bránu - 192.168.11.0 a dostane sa do smerovača B. Ten vo svojej tabuľke nájde záznam, ktorý hovorí, že podsieť 192.168.16.0/24 je dostupná prostredníctvom rozhrania Serial0/1 a next-hop adresa (adresa nasledujúceho smerovača) je 192.168.12.2. Smerovač teda prepošle paket na toto rozhranie. Obdobná situácia by sa mohla opakovať viackrát, až sa paket dostane do posledného smerovača. Ten v tabuľke nájde záznam udávajúci, že cieľové zariadenie je priamo pripojené a jednoducho odovzdá paket na daný port rozhrania a paket je doručený do cieľa.

Typicky existuje obrovské množstvo rôznych ciest, ktoré nás dovedú do jedného cieľa. Úlohou smerovačov je vybrať, pokiaľ možno, tú najlepšiu. Z pohľadu teórie grafov sa jedná o optimalizačnú úlohu pre hľadanie optimálnej cesty. Faktorom výberu však nie je len vzdialenosť do cieľa. Smerovače musia dynamicky reagovať tiež na nedostupnosť alebo zahltenosť niektorých liniek a ďalšie situácie. Je to rozsiahla a komplikovaná úloha, ktorú vykonávajú tzv. smerovacie protokoly (napr. RIP, OSPF, IGRP [20]). Pomocou nich si smerovače medzi sebou priebežne vymieňajú informácie o topológii a aktuálnom stave siete. Na základe týchto informácií je možné vypočítať vhodnú trasu.

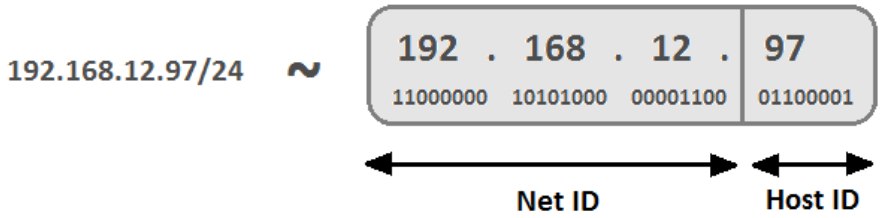


Obrázok 2.4: Ukážka smerovacej tabuľky a odpovedajúcej topológie

2.4 Adresovanie v protokole IP

Pre správnu funkčnosť smerovania a doručovania dát do cieľa, potrebujeme poznať adresu cieľového zariadenia. Ako už bolo spomenuté, toto adresovanie sa na sieťovej vrstve vykonáva pomocou IP adries. V tejto časti je predstavený bližší pohľad na adresovanie v protokole IPv4 a IPv6, ich odlišnosti a charakteristické vlastnosti.

Adresa IP vo verzii 4 je logicky rozdelená na 2 časti – prvá časť identifikuje sieť, v ktorej sa počítač nachádza (Net ID) a druhá časť identifikuje konkrétny počítač v danej sieti (Host ID). Počet bitov, ktoré reprezentujú sieť vyjadruje maska siete a môžeme ju zapisovať vo formáte prefixu (napr. 192.168.12.97/24 znamená, že prvých 24 bitov predstavuje Net ID). Situácia je znázornená na obrázku 2.5.



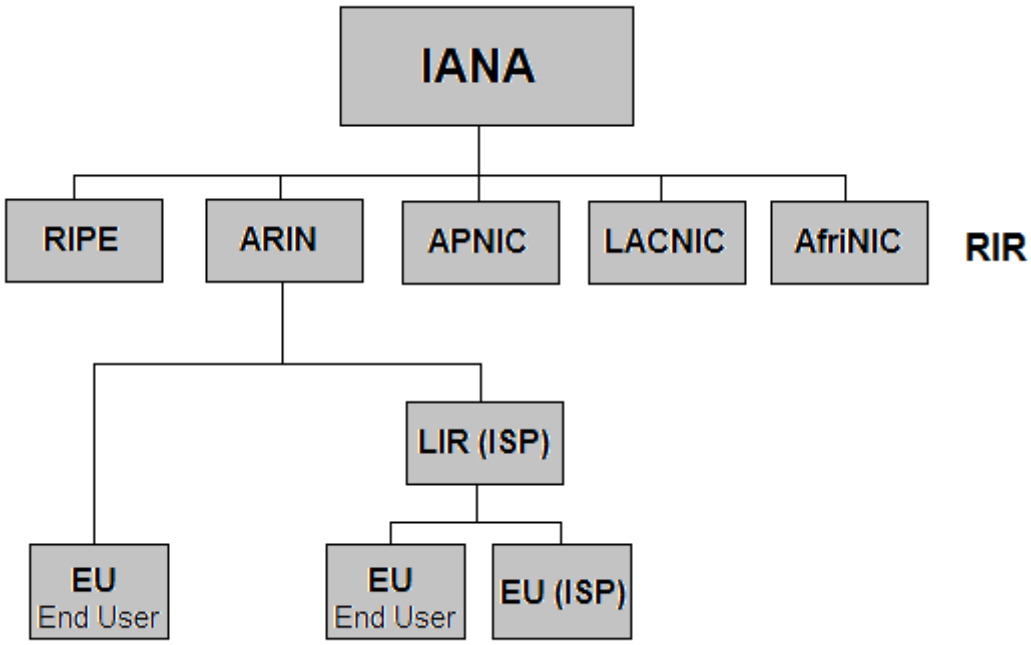
Obrázok 2.5: Znázornenie 2 úrovní IPv4 adresy

Podľa veľkosti masky siete boli IP adresy spočiatku rozdelené do tried (tabuľka 2.1). Tento spôsob však neponúka dostatočnú škálovateľnosť pri prideľovaní blokov adries a dochádzalo k ich neefektívnemu využívaniu a plytvaniu adresným priestorom. Preto sa prešlo na beztriedne adresovanie a smerovanie (CIRD, Classless Inter-domain Routing) [5], kedy môže mať sieťová časť IP adresy ľubovoľný počet bitov.

Trieda adries	Začiatok adresy		Prefix siete
Trieda A	0	1-126	/8 = 255.0.0.0
Trieda B	10	128-191	/16 = 255.255.0.0
Trieda C	110	192-223	/24 = 255.255.255.0
Trieda D	1110	224-239	multicast
Trieda E	1111	240-254	Rezervované pre experimentálne účely

Tabuľka 2.1: Rozdelenie IPv4 adries do tried

Prideľovanie blokov IP adries má na najvyššej úrovni na starosti organizácia IANA (Internet Assigned Numbers Authority). Tá spravuje celý adresný priestor a distribuuje adresné bloky regionálnym registrátorom (RIR – Regional Internet Registry), ktorých je v súčasnosti 5 – RIPE, ARIN, APNIC, LACNIC, AfriNIC. Každý z nich je zodpovedný za jeden rozsiahly geografický región a má vo svojej kompetencii ďalšie rozdeľovanie alokovaného adresného priestoru. Na nižšej úrovni sa potom nachádzajú lokálni registrátori (LIR – Local Internet Registry). Tí zvyčajne predstavujú už konkrétnych poskytovateľov internetu (ISP – Internet Service Provider). Ich zákazníkmi sú koncoví užívatelia (prípadne ďalší ISP), ktorým ISP umožňujú pripojiť sa do siete internet. Celú túto štruktúru prehľadne ilustruje obrázok 2.6. Od alokačnej politiky jednotlivých autorít do veľkej miery závisí efektivita rozdelenia a využitia adresných blokov.



Obrázok 2.6: Hierarchia prideľovania blokov IP adries

Pri novom protokole bolo nevyhnutné poučiť sa z predchádzajúcich nedostatkov a tiež z charakteristických rysov vývoja internetu. Štvornásobné zvýšenie šírky adresy prináša oproti IPv4 takmer neobmedzené množstvo adries. Je teda potrebné zvýšiť efektivitu spravovania tohto obrovského adresného priestoru. To je dosiahnuté hlavne zdokonalením hierarchickej štruktúry prideľovania adries, čo nám dovolí lepšiu agregáciu adries a tiež zabráni neúmernému nárastu veľkosti smerovacích tabuliek. Triedne rozdelenie adries bolo vo verzii 6 úplne odstránené.

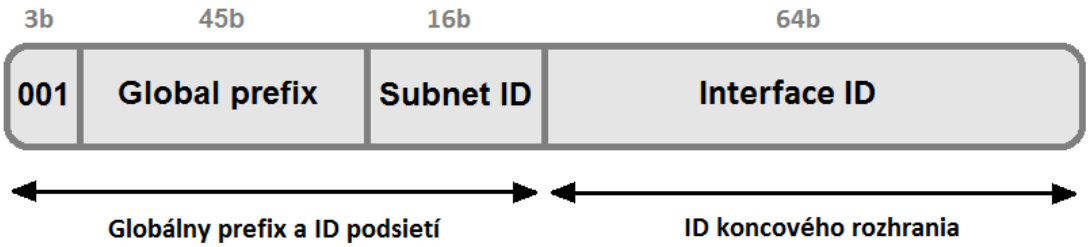
Internetové adresy verzie 6 delíme na niekoľko skupín, ktoré odlišujeme podľa prvých bitov adresy. Aktuálne rozdelenie zobrazuje tabuľka 2.2. Prvý riadok ukazuje nešpecifikovanú adresu 0:0:0:0:0:0:0:0. Táto adresa nesmie byť nikdy pridelená žiadnemu uzlu a predstavuje absenciu adresy. Príklad jej využitia je v zariadení, ktoré ešte nezistilo svoju vlastnú IP adresu.

Typ adresy	Binárny prefix	IPv6 notácia
Unspecified	00...0 (128 bits)	::/128
Loopback	00...1 (128 bits)	::1/128
Multicast	11111111	FF00::/8
Link-local unicast	1111111010	FE80::/10
Global unicast	001	2000::/3

Tabuľka 2.2: Rozdelenie typov IPv6 adries

Obvykle je adresa IPv6 rozdelená na 3 časti – globálny prefix (n bitov), ID podsiete (m bitov), ID koncového rozhrania (128 – n – m). Táto štruktúra pripomína použitie podsietí v IPv4. Avšak

adresovanie v IPv6 bolo navrhnuté s cieľom vyššej agregácie. Toto rozdelenie IP adresy vidíme na obr. 2.7. Obrázok znázorňuje adresu typu global unicast, ktorá sa vyskytuje najčastejšie. Globálny prefix je typicky hierarchicky štrukturovaná hodnota, ktorá odpovedá adrese siete. Utvárajú ju regionálni a lokálni registrátori a poskytovatelia internetu. Pole „Subnet ID“ má v plnej správe administrátor siete a môže vďaka nej vytvárať ďalšie podsiete, podobne ako pri protokole IPv4. Typicky platí, že globálny prefix spolu s ID podsiete tvoria prvých 64 bitov adresy. Nižších 64 bitov pripadá na ID zariadenia a z pohľadu smerovania zvyčajne táto časť nie je podstatná. Identifikátor zariadenia musí byť jedinečný v rámci svojej siete, pretože jednoznačne určuje konkrétne zariadenie v sieti, podobne ako MAC adresa. Existuje viacero prístupov, na základe ktorých môžeme vygenerovať túto časť IP adresy. Typicky sa k tejto úlohe využíva práve 48 bitová MAC adresa zariadenia, ktorá sa namapuje na 64 bitov. To nám zaručuje dokonca globálnu unikátnosť. Táto technika však obsahuje isté bezpečnostné riziká. Adresa zostáva statická a veľmi jednoducho dokážeme identifikovať a sledovať užívateľov. Z toho dôvodu bolo navrhnuté náhodné generovanie ID rozhrania, ktoré sa navyše po čase mení a generuje sa znovu. Tento prístup pomocou dočasných adries zabezpečí anonymitu koncového zariadenia.



Obrázok 2.7: Typické hierarchické rozdelenie adresy IPv6

Adresy IPv6 sú taktiež veľmi často reprezentované v prefixovom formáte (napr. 12AB:0:0:CD30::/60) Maska v tomto prípade predstavuje globálny prefix. Protokol IP verzie 6 prináša so sebou aj niektoré nové typy adries. Jednou z nich je IPv4 kompatibilná adresa. Ako názov napovedá, tento typ pomáha zabezpečiť súčasné použitie oboch protokolov a ich vzájomnú kompatibilitu. Jej formát pripomína akýsi mix oboch verzií. Spodných 32 bitov adresy predstavuje pôvodnú IPv4 adresu a je zapísaná v klasickom bodkovom formáte, pokým zvyšok adresy je vo formáte typickom pre IPv6. Vo výsledku má teda adresa tvar:
`xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:ddd.ddd.ddd.ddd`.

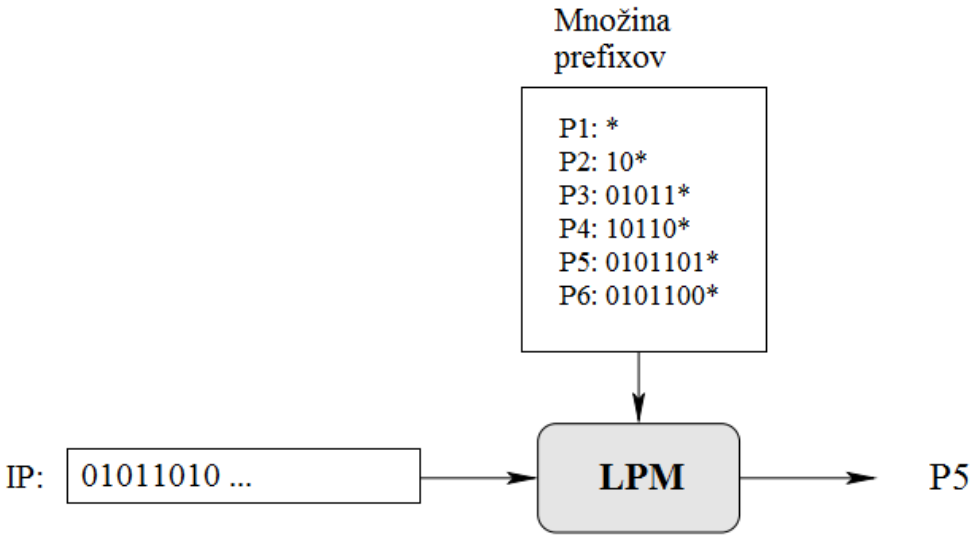
V súčasnosti prideľuje organizácia IANA regionálnym registrátorom (RIR) bloky IPv6 adries veľkosti /23, ktoré začínajú prefixom 2000::/3 (binárne 001). Na základe týchto prvých 3 bitov IP adresy dokážeme okamžite určiť, že sa jedná o adresu typu gloabal unicast. RIR spravujú alokované bloky vo svojej kompetencii a ďalej ich poskytujú lokálnym registrátorom (LIR a ISP). Minimálna alokačná jednotka je v tomto prípade /32. Organizácie môžu žiadať o alokácie až po bloky veľkosti /29. Žiadosť o väčšie bloky je potrebné podložiť dokumentáciou popisujúcou opodstatnenosť tohto požiadavku. Koncoví užívatelia – menšie ISP a organizácie, majú pridelené adresy s prefixom /48, ktoré dostanú od svojho lokálneho registrátora. Keďže všetky IPv6 siete majú k dispozícii 64 bitov na adresovanie siete, zostáva v tomto bloku ešte 16 bitov na adresáciu podsietí. Teoreticky teda každý pridelený blok veľkosti /48 umožňuje adresovať 65 536 LAN sietí. Ďalšou typickou alokačnou jednotkou je /56, ktorá sa prideľuje malým firmám a domácnostiam (dovoľuje vytvoriť až 256

podsietí). Blok /64 sa používa v prípade, ak je známe, že je potrebná jedna a práve jedna podsieť (tá môže obsahovať až 2^{64} zariadení). Je prípustné alokovať tiež adresu /128 a to vtedy, ak je definitívne, že jedno a práve jedno zariadenie bude pripojené. Uvedená alokačná politika ovplyvňuje charakteristické rysy súčasných a budúcich smerovacích tabuliek.

2.5 Vyhľadanie najdlhšieho prefixu

V predchádzajúcich častiach sme si stručne priblížili problematiku adresovania a smerovania v IP sieťach. Kľúčovú úlohu v tomto procese zohrávajú smerovače, ktoré vyberajú pre naše dáta optimálnu cestu a zabezpečujú doručovanie paketov do cieľového zariadenia. Primárnou funkciou smerovačov je vyhľadanie adresy nasledujúceho uzlu a preposlanie paketu na príslušné výstupné rozhranie. K tomu sa zvyčajne pridružujú mnohé ďalšie úkony, avšak časovo kritickou operáciou je vyhľadanie IP adresy v smerovacej tabuľke. Práve k tomu sa využívajú algoritmy pre vyhľadávanie najdlhšieho zhodného prefixu - LPM.

Každý LPM algoritmus má na vstupe množinu prefixov, v ktorej bude prebiehať vyhľadávanie. Táto množina IP prefixov je vytvorená na základe smerovacej tabuľky a je uložená v pamäti smerovača. Ďalej máme na vstupe cieľovú IP adresu smerovaného paketu. Úlohou algoritmu je vybrať z množiny taký prefix, ktorý sa zhoduje so vstupnou IP adresou. Pritom však nestačí vyhľadať prvú zhodu, ale je nutné prehľadať celú prefixovú množinu a vybrať prefix, ktorý sa zhoduje v najvyššom počte bitov. Výstupom algoritmu je potom prefix, ktorý je najdlhší, a teda najšpecifickejší. Algoritmus následne oznámi výsledok a vykoná sa akcia, ktorá je s daným záznamom spojená – typicky preposlanie paketu na next-hop adresu. Algoritmus by sa mal snažiť riešiť túto úlohu v čo najkratšom čase a s minimálnou pamäťovou zložitosťou. Uvedený základný princíp LPM je ilustrovaný na obrázku 2.8.



Obrázok 2.8: Obecný princíp LPM

Pri práci a vyhľadávaní pracujeme s prefixami v binárnej podobe, teda napr. IPv4 prefix 208.12.16/20 je binárne reprezentovaný 11010000 00001100 0001*. Tento hviezdičkový zápis je pre LPM typický a znamená, že nižšie bity už nie sú súčasťou prefixu. Uvedený prefix teda agreguje všetky možné adresy, ktoré majú prvých 20 bitov zhodných (11010000 00001100 0001). Výber najdlhšieho zhodného prefixu je preto podstatný a pomáha nám urýchľovať postup smerovania a výber optimálnej cesty. Po nájdení najdlhšieho zhodného prefixu bude teda paket preposlaný do siete, ktorá je najbližšie cieľovému zariadeniu (na základe dostupných informácií v danej smerovacej tabuľke). Týmto spôsobom každý smerovač určí najlepší možný záznam a paket sa približuje do cieľového zariadenia.

Základný princíp LPM sa zdá byť jednoduchý, avšak situácia v reálnom nasadení býva omnoho komplikovanejšia a musíme prekonávať mnohé problémy. Vyhľadanie prefixu sa musí vykonávať nad každým paketom zvlášť a s neustálym zvyšovaním rýchlostí liniek sa čas vyhradený na túto operáciu skracuje. Pre dosiahnutie priepustnosti na úrovni 100 Gbps je potrebné vykonať až 160 miliónov vyhľadaní za sekundu. Z toho vyplýva, že čas na jedno vyhľadanie je maximálne 6 ns. Operácia LPM sa teda skutočne stáva veľmi časovo kritickou úlohou.

Zvyšovanie rýchlostí však nie je jediný problém. Rovnako narastajú tiež smerovacie tabuľky a odpovedajúce prefixové množiny, v ktorých prebieha vyhľadávanie. Smerovacie tabuľky typicky obsahujú stovky, tisícky alebo až stovky tisíc záznamov. Priestor, ktorý musíme za tak krátku dobu prehľadať, je teda značne rozsiahly. Všetky záznamy musia byť vo vhodnej reprezentácii uložené v pamäti, čo kladie nároky na priestorovú zložitosť algoritmu.

Ďalším problémom v oblasti LPM je prechod na nový IP protokol. V prípade verzii IPv4 boli všetky prichádzajúce IP adresy veľkosti 32 bitov a ich prefixy mali dĺžku 0-32 bitov. S príchodom protokolu IPv6 však veľkosť IP adresy narástla až na 128 bitov. Prefixy protokolu IPv6 majú teoreticky tiež veľkosť až 128 bitov, typicky sa však v oblasti smerovania využíva len vrchných 64 bitov. Napriek tomu je tento rozdiel veľmi podstatný, keďže sa mnohonásobne zvýšil počet bitov, ktoré musíme porovnávať na zhodu. To samozrejme so sebou zase prináša dodatočné nároky na pamäť a výkon algoritmov.

Smerovače teda v takýchto podmienkach nemajú jednoduchú úlohu a musia sa s uvedenými faktormi vyrovnať a poskytovať za všetkých okolností dostatočnú priepustnosť. Efektívne techniky LPM sú preto nevyhnutné pre zabezpečenie kontinuálneho vývoja a zvyšovanie prenosových rýchlostí liniek. Pritom sme limitovaní veľkosťou a latenciou pamätí, rýchlosťou procesora a podobne. Logickou požiadavkou je čo najvyššia rýchlosť spracovania každého paketu. Zároveň sa však musíme snažiť minimalizovať pamäťovú náročnosť riešenia.

LPM je obecný princíp a využíva sa nielen na vyhľadávanie prefixov pri smerovaní, ale má uplatnenie tiež pri klasifikácii paketov, filtrovaní dát a v iných oblastiach. V oblasti LPM existuje veľké množstvo algoritmov postavených na rôznych princípoch [17] [22]. Zvyčajne sú efektívne a dobre optimalizované pre použitie s adresami IPv4. Tieto známe prístupy sú však dnes už nedostatočné a v kontexte protokolu IPv6 zvyčajne zlyhávajú. Je potrebné sa na tento nedostatok zamerať a hľadať nové cesty a možnosti. Práve preto som si za jeden z cieľov mojej práce stanovil návrh nového a efektívneho algoritmu pre prácu s týmito adresami.

3 Popis LPM algoritmov

V tejto kapitole postupne predstavím viaceré známe algoritmy na vyhľadávanie najdlhšieho zhodného prefixu. Popíšem ich významné vlastnosti a základné dátové štruktúry, na ktorých sú postavené. Na jednoduchých príkladoch bude ilustrované uloženie a kódovanie dát, ako tiež samotný postup vyhľadávania.

Aby sme vedeli porovnať výkon jednotlivých algoritmov, používajú sa najčastejšie nasledujúce typy metrík:

- **Rýchlosť vyhľadania**. Šírka pásma používaných liniek sa neustále zvyšuje. Tým pádom sa maximálny čas vyhradený pre spracovanie jedného paketu musí znižovať. Rýchlosť akou vyhľadáme správny prefix je pritom zásadná.
- **Pamäťové nároky**. Sú podstatné pre možnosť využitia cache a SRAM pamätí. Malá pamäť znamená nízku spotrebu a krátku dobu prístupu, čo zase zvyšuje tiež rýchlosť algoritmu.
- **Čas aktualizácie.** V niektorých prípadoch môže byť nutné vykonať až stovky aktualizácií za sekundu, aby sme sa vyhli nestabilite smerovania. Aktualizácie by nemali interferovať s klasickými vyhľadávacími operáciami.
- **Škálovateľnosť**. Smerovacie tabuľky sa každým rokom zväčšujú. Je požadované aby bol algoritmus pripravený a schopný zvládnuť aj tabuľky o veľkosti stoviek tisíc záznamov a viac.
- **Flexibilita implementácie**. Niektoré algoritmy poskytujú lepšiu flexibilitu, pretože je možné ich efektívne implementovať do ASIC, sieťového procesoru, generického procesoru a podobne.

Naivný prístup k vyhľadávaniu je zoradiť databázu do jednoduchého zoznamu a prehľadávať postupne všetky položky až dokým nenarazíme na zhodu. V prípade LPM navyše nestačí vyhľadať prvú zhodu, ale museli by sme vždy prejsť celý zoznam a vybrať najlepšiu z nich. Toto riešenie by fungovalo pre malé množiny, ale rýchlosť vyhľadania klesá s každou pridanou položkou a je neprijateľná.

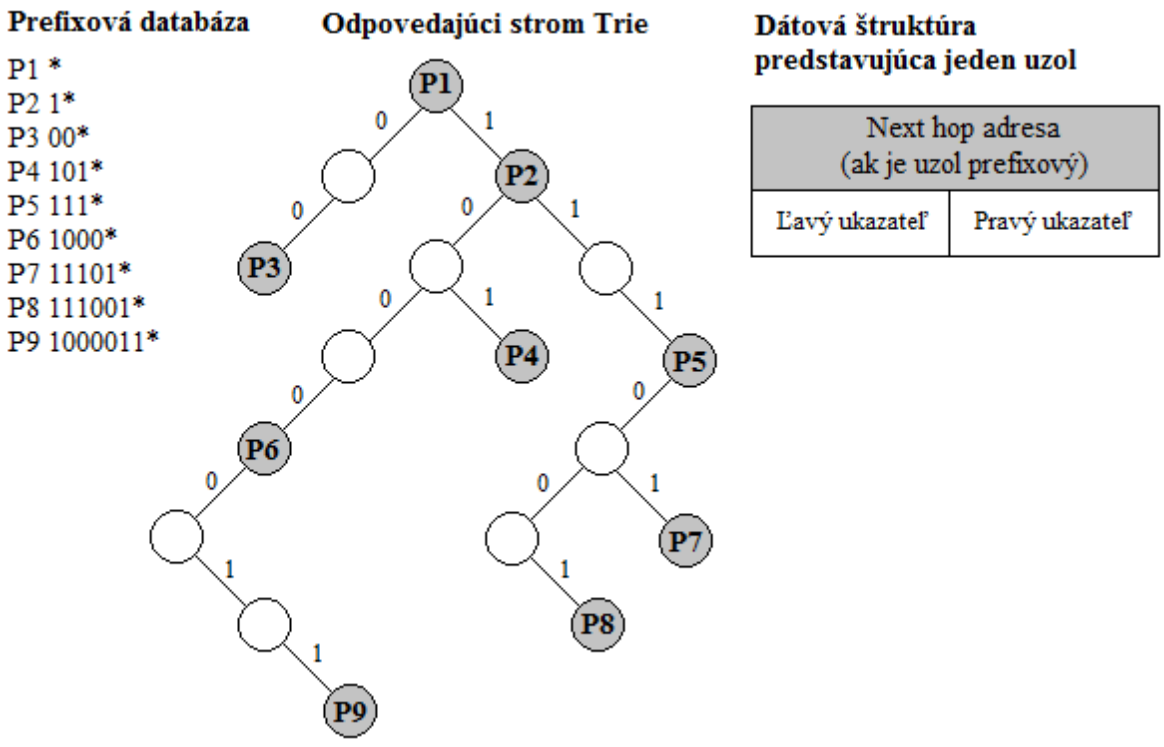
Na podobnej jednoduchej myšlienke pracujú asociatívne pamäte typu TCAM. Je to hardwarové riešenie, ktoré dokáže získať výsledok vyhľadania v jednom cykle, avšak priepustnosť je limitovaná relatívne nízkou rýchlosťou týchto pamätí. Nevýhodou je tiež pomerne malá kapacita a veľmi vysoká spotreba energie. Navyše sú tieto špecializované pamäte drahé. Preto bolo navrhnutých mnoho algoritmických riešení, ktoré sú univerzálne a snažia sa uvedené nedostatky minimalizovať.

Cieľom LPM algoritmov je usporiadať databázu prefixov do takej dátovej štruktúry, ktorá je optimalizovaná pre vyhľadávanie. Táto transformácia sa vykonáva jednorazovo – pri modifikácii vstupnej množiny. Samotná operácia vyhľadania potom pracuje zásadne nad optimalizovanou reprezentáciou. Množstvo algoritmov využíva na uloženie prefixov stromovú štruktúru. Výhodou je, že dobre pracuje s prehľadávaným priestorom, ktorý sa pri vyhľadávaní rýchlo zmenšuje. V prípade vyváženého binárneho stromu sa prehľadávaný priestor v každom kroku obmedzí na polovicu.

3.1 Trie

Algoritmus Trie [2] je postavený na myšlienke využiť jednoduchý binárny strom pre uloženie prefixov priamo vo svojej konštrukcii. Každý uzol predstavuje 1 možný prefix a môže mať nula až dvoch následníkov. Pri kódovaní prefixu začíname v koreni stromu a prechádzame prefix po bitoch počínajúc MSB (most significant bit). Ak sa v adrese nachádza 0, posunieme sa na ľavého následníka, ak 1, posunieme sa na pravého následníka. Ak potrebný následník ešte neexistuje, tak si ho vytvoríme. Uzol, v ktorom prefix končí, označíme ako „prefixový“ a doplníme mu potrebné údaje (next-hop). Ostatné uzly slúžia len ako prechodné. Týmto spôsobom zakódujeme celú množinu prefixov, čím nám vznikne vyhľadávací strom. Koreň stromu reprezentuje generický prefix *, ktorý odpovedá ľubovoľnej IP adrese. Príklad databázy a odpovedajúci strom Trie je na obrázku 3.1.

Vyhľadávanie je veľmi podobné princípu tvorby stromu. Vstupnú adresu zase prechádzame po jednotlivých bitoch a rovnakým spôsobom sa pohybujeme v strome. Počas priechodu stromom si zaznamenávame posledný platný prefix, na ktorý sme narazili. Vyhľadávanie končí, ak v požadovanom smere neexistuje následník alebo sme spracovali všetky vstupné bity. Výsledkom je posledný nájdený prefix a jeho next-hop hodnota.



Obrázok 3.1: Ukážka stromu vytvoreného algoritmom Trie

Tento algoritmus ukazuje základné výhody použitia stromovej štruktúry. Je triviálny a poskytuje jednoduchú aktualizáciu vstupnej množiny. Výhodou Trie a odvodených algoritmov je agregácia prefixov v strome. Na uvedenom obrázku je prefix P5 obecnejší a agreguje špecifickejšie prefixy P7 a P8. Veľká časť prefixov je teda zdieľaná a je uložená len jedenkrát. V porovnaní s naivným prístupom sa takto odstraňuje veľké množstvo redundantných informácií. Nevýhodou algoritmu je nízka rýchlosť vyhľadávania, ktorá vyplýva z vysokého počtu prístupov do pamäte. Pri vyhľadaní sa počet potrebných krokov rovná výške stromu (dĺžka prefixu). Tá je v najhoršom prípade 32 pre IPv4, resp. 128 pre IPv6. Taktiež celkové pamäťové nároky sú vysoké. Veľký počet ukazateľov a prechodných uzlov zaberajú priestor na úkor užitočných informácií. Algoritmus Trie je v reálnych podmienkach prakticky nepoužiteľný. Na rozvíjaní tejto základnej myšlienky je však postavených množstvo sofistikovanejších riešení, ktoré je možné využiť vo vysokorýchlostných zariadeniach [8].

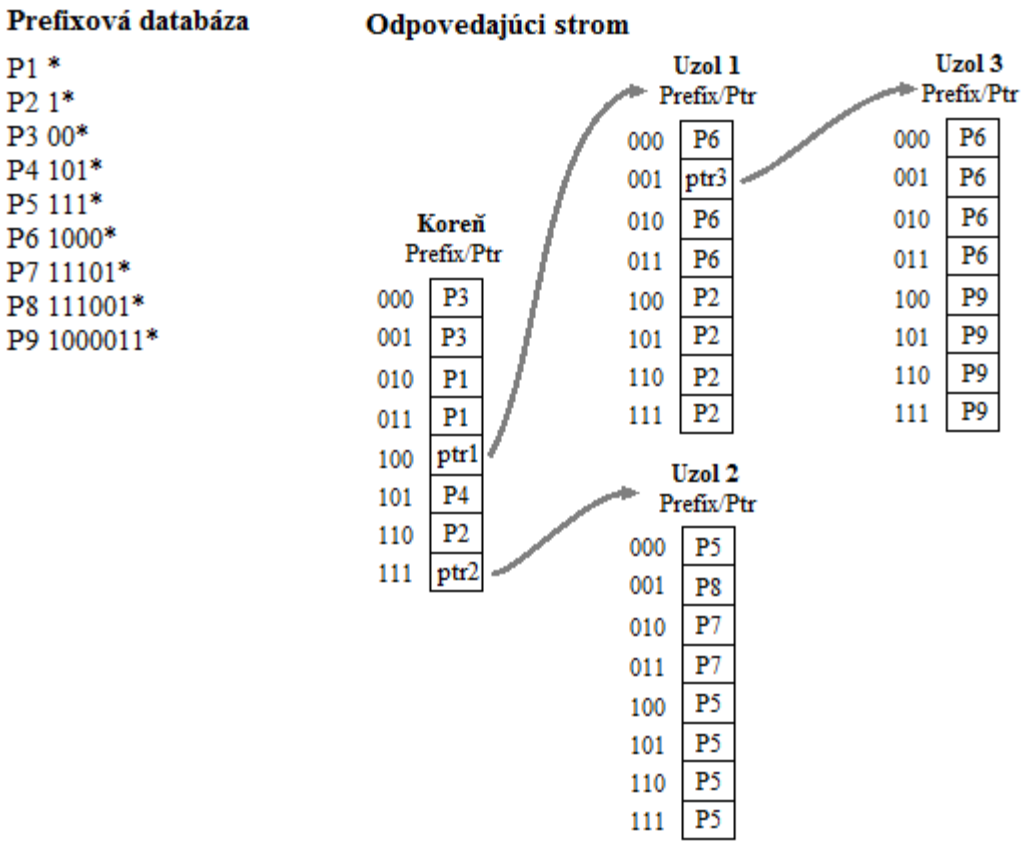
3.2 Controlled Prefix Expansion

S cieľom znížiť počet prístupov do pamäte vznikli algoritmy, ktoré spracovávajú viac bitov naraz v každom takte. Na tomto princípe pracuje tiež algoritmus Controlled Prefix Expansion (CPE), ktorý bol predstavený v [21]. Takéto metódy nazývame multibit a počet bitov spravovaných v jednom takte môžeme nastaviť pomocou parametra n (strieda). Nie všetky prefixy v množine však majú dĺžku, ktorá odpovedá násobku hodnoty n. Pri týchto prefixoch musíme vykonať ich expanziu na plnú dĺžku striedy. Ak je hodnota n=3, tak napr. prefix 1* expanduje na prefixy 100, 101, 110, 111.

Jeden uzol teda ukladá n bitov prefixu, čo znamená 2^n položiek. Pri každej položke musíme zaznamenať, či obsahuje platný prefix (next-hop adresu) a ukazateľ na následníka. To vedie na vysokú pamäťovú náročnosť, ktorú môžeme čiastočne znížiť optimalizáciou „leaf pushing“. Jeho myšlienkou je neukladať pri každej položke prefix a zároveň následníka, ale len jednu z týchto informácií. Prípadne, že boli v pôvodnom zázname obe, presúvame informácie o prefixe do nasledujúceho uzlu. Príklad stromu vystavaného algoritmom CPE s optimalizáciou leaf pushing je na obrázku 3.2.

Pri vyhľadávaní použijeme v každom kroku trojicu bitov adresy ako index do tabuľky. Ak sa na tomto mieste nachádza ukazateľ, pokračujeme do nasledujúceho uzlu. V prípade, že je tu prefix, vyhľadávanie končí a odovzdá next-hop informáciu. Vyhľadávanie môže byť ukončené tiež, ak je na danom miesto nulový ukazateľ. V tom prípade neexistuje pre danú adresu žiadny platný prefix.

Algoritmus zlepšuje rýchlosť vyhľadávania a znižuje počet prístupov do pamäte, ktoré sú W/n, kde W je maximálna dĺžka prefixu a n je parameter striedy. Pamäťová náročnosť však zostáva príliš vysoká i s využitím leaf pushing, kde každý uzol potrebuje 2^n ukazateľov.



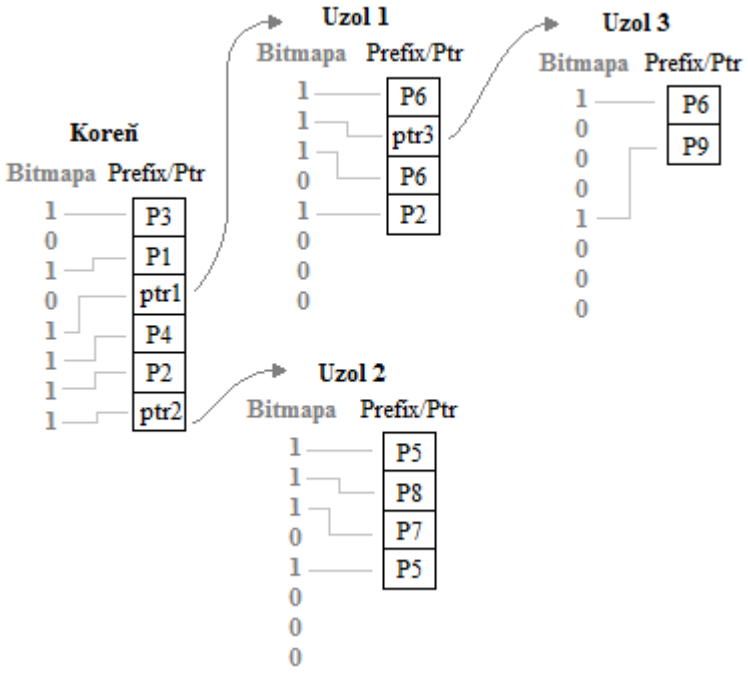
Obrázok 3.2: Strom vytvorený algoritmom CPE (n=3) s optimalizáciou Leaf Pushing

3.3 Lulea Compressed Tries

Nevýhodou predchádzajúceho algoritmu je veľký počet redundantných údajov. Tú sa snaží odstrániť algoritmus Lulea Compressed Tries [4].

Vychádza z konceptu CPE s použitím leaf pushing, ktorý ďalej rozvíja. Na odstránenie redundantných položiek v tabuľke používame bitmapu, ktorá má veľkosť 2^n bitov, kde n je použitá strieda algoritmu. Ako sme videli, obsahujú tabuľky v uzle CPE opakujúce sa hodnoty za sebou. Lulea zachová vždy len prvú z nich a na danú pozíciu bitmapy zapíše hodnotu 1. Bezprostredne nasledujúce zhodné údaje odstráni a v bitmape sú označí nulou. Pre ilustráciu použijeme strom z obrázku 3.2 a aplikujeme naň algoritmus Lulea. Výsledok a ušetrenie pamäte odstránením redundantných údajov zobrazuje obr. 3.3.

Vyhľadávanie prebieha veľmi podobne ako v predchádzajúcom algoritme. Segment vstupnej adresy však tentokrát použijeme ako index do bitmapy. Spočítame počet jedničiek od začiatku bitmapy až po daný index. Až táto vypočítaná hodnota nám slúži ako index do tabuľky. Ak sa na danej pozícii vyskytuje ukazateľ, pokračuje vo vyhľadávaní. Ak je tu nulový odkaz, končí vyhľadávanie neúspešne (neexistuje platný prefix pre adresu). V prípade, že narazíme na prefix, je vyhľadanie úspešné.



Obrázok 3.3: Ukážka optimalizácie stromu pomocou algoritmu Lulea

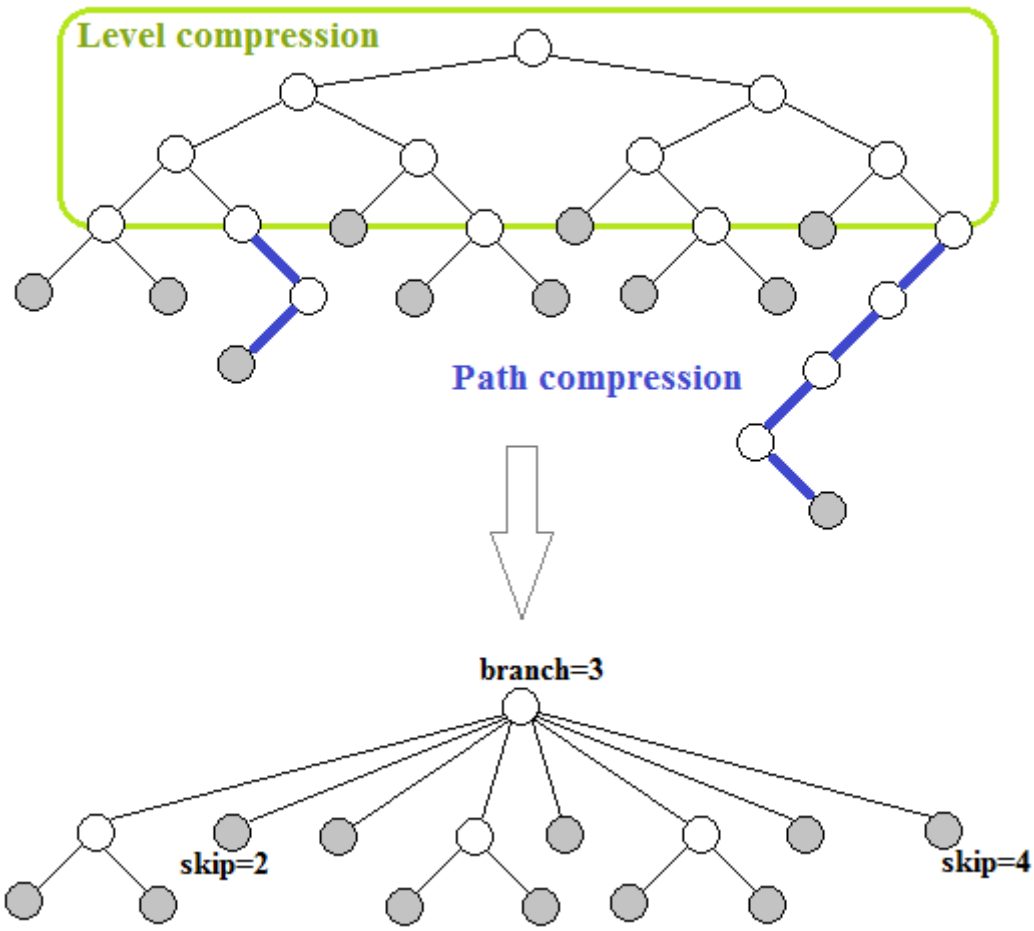
Jeden uzol algoritmus Lulea obsahuje v najhoršom prípade 2^n ukazateľov a navyše bitmapu, ktorá má 2^n bitov. Môže sa zdať, že je teda na tom dokonca ešte horšie ako algoritmus CPE. Takéto hraničné situácie sa však prakticky nevyskytujú a ušetrená pamäť je zvyčajne veľmi výrazná. Čo sa týka rýchlosti vyhľadania, tá zostáva nezmenená. Hlavnou nevýhodou však je, že nepodporuje inkrementálne aktualizácie vstupnej množiny.

3.4 Level Compression Trie

Algoritmus nazývaný Level Compression Trie (LC-Trie) bol navrhnutý v [12] a optimalizuje binárny strom dvoma spôsobmi.

Prvou optimalizáciou je technika „path compression“, ktorá vyhľadá nevetvené cesty v strome a následne ich vynechá. Počet odstránených uzlov cesty udáva hodnota „skip value“. Túto hodnotu uložíme do prvého uzlu, ktorý nasleduje za odstráneným segmentom. Zároveň si v ňom poznačíme bitmapu preskočeného úseku.

Druhá optimalizácia je nazývaná „level compression“. Tá využíva výhody multibit algoritmov. Multibit metódy zvyšujú výkon, ale objavujú sa redundantné údaje. Preto je výhodné použiť multibit len na podstromy, ktoré sú plne vyvážené. Ak máme plne vyvážený strom s výškou k, môžeme použiť tento typ kompresie a všetky jeho uzly nahradiť jediným uzlom stupňa 2^k – teda bude mať 2^k následníkov. Túto hodnotu k uložíme do uzla ako vetviaci pomer – „branch“. Ukážka použitia oboch typov optimalizácií je znázornená na obrázku 3.4.



Obrázok 3.4: Ukážka tvorby stromu LC-Trie

Vyhľadávanie musí rešpektovať uvedené optimalizácie v strome. V každom uzle sa pozrieme na hodnotu „skip value“ a „branch“. Ak je skip hodnota rôzna od nuly, tak preskočíme daný počet bitov adresy a podľa nasledujúceho bitu pokračujeme do ďalšieho uzlu. Ešte predtým však musíme skontrolovať, či sa preskočená bitmapa stromu zhodovala s preskočenou časťou adresy. Ak je hodnota branch (k) rôzna od 1, použijeme k bitov adresy ako hodnotu indexu a vo vyhľadávaní pokračujeme v následníkovi na tomto indexe.

Tvorba stromu, s ním spojené operácie a vyhľadávanie sú pomerne komplikované záležitosti, ktorých bližší popis sa nachádza v [12]. Nevýhodou algoritmu je komplikovaná hardwarová implementácia. Taktiež inkrementálna aktualizácia je veľmi obtiažna. Zato však poskytuje dobré výsledky z pohľadu správy pamäti a rýchlosti.

3.5 Tree Bitmap

Algoritmus predstavený pod názvom Tree Bitmap [3] patrí medzi multibit metódy s v využitím bitmapovej kompresie. Jeho prednosti sa ukázali v čase uvedenia ako veľmi perspektívne a poskytuje rozumné vyváženie výkonnostných kritérií – dobrá rýchlosť vyhľadania, pamäťová náročnosť a schopnosť rýchlej aktualizácie.

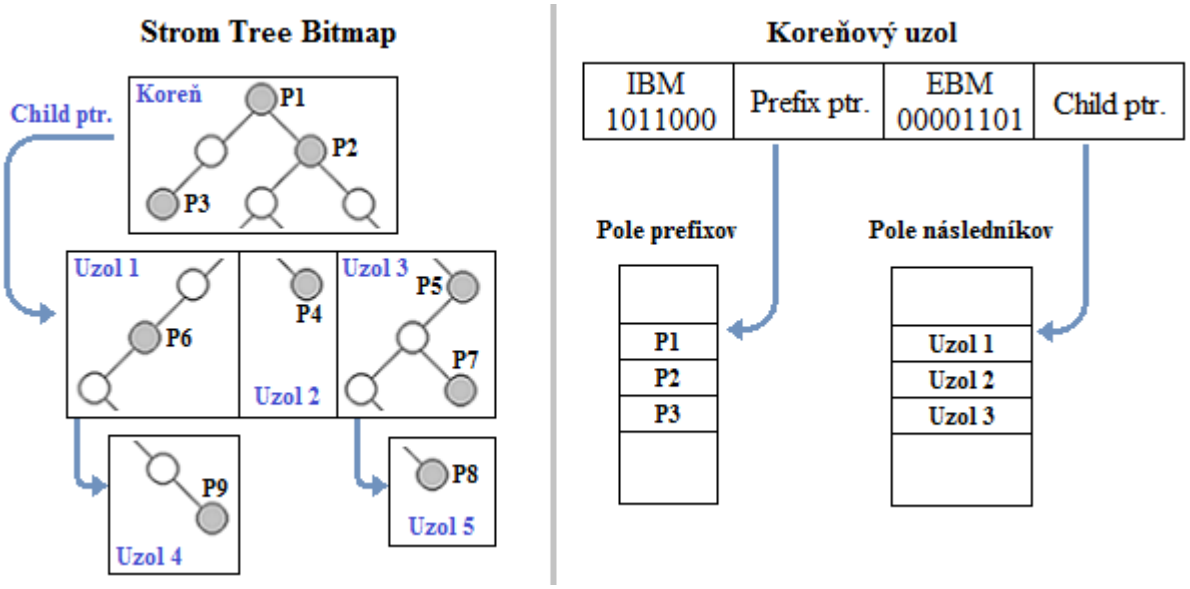
Každý uzol má pevnú veľkosť, ktorá závisí od zadaného kroku n – počet bitov adresy spracovaných v jednom takte. Jeden uzol obsahuje:

- Internú bitmapu (IBM)
- Externú bitmapu (EBM)
- Odkaz do poľa s platným prefixmi
- Odkaz do poľa s následníkmi

Interná bitmapa kóduje pozície, na ktorých sa nachádzal platný prefix. Na jej vytvorenie sa uzly jednoduchej Trie prechádzajú v breadth-first poradí. Ak uzol obsahuje prefix, zapíšeme 1, inak 0. Takto nám vznikne bitmapa veľkosti 2^n-1 bitov. Externá bitmapa zase uchováva pozície, na ktorých sa nachádza nasledovník. Prejdeme n-tú hladinu uzlov Trie a na mieste, kde existuje následník zapíšeme 1, na zvyšných 0. Tak vznikne externá bitmapa veľkosti 2^n.

Všetky platné prefixy daného uzlu sú uložené v poli bezprostredne za sebou. Vystačíme si potom s odkazom na prvý z nich a pomocou bitmapy vieme spočítať offset na nasledujúce. Rovnakým spôsobom sa pracuje so synovskými uzlami, ktoré sú uložené v druhom poli. Tento spôsob uloženia informácií šetrí veľké množstvo odkazov a znižuje pamäťové nároky. Ukážka vytvorenej dátovej štruktúry metódou Tree Bitmap sa nachádza na obr. 3.5.

Pri vyhľadávaní použijeme v každom uzle n bitov vstupnej adresy ako index do externej bitmapy. Ak je na tomto mieste hodnota 1, znamená to, že v danom smere sa nachádza ďalší uzol. Skontrolujeme teda platné prefixy daného uzlu a presunieme sa do synovského uzla. Potrebný offset spočítame ako počet jedničiek od začiatku po náš index. Vyhľadávanie končí, ak spracujeme všetky bity vstupnej adresy alebo sa dostaneme na koniec stromu (hodnota 0 v externej bitmape).



Obrázok 3.5: Dátové štruktúry algoritmu Tree Bitmap (n=3)

V porovnaní s algoritmom Lulea, ktorý tiež využíva bitmapu, je na tom Tree Bitmap výkonnostne omnoho lepšie. Vyhľadanie prebehne v najhoršom prípade za W/n krokov, kde n je maximálna dĺžka prefixu a n je použitá strieda. Navyše umožňuje jednoduchú a rýchlu aktualizáciu. Tree Bitmap vsadil na vyššiu náročnosť spracovania uzlu, zato však získame nižší počet prístupov do pamäte. Pamäťové nároky na uloženie jedného uzlu rastú exponenciálne s použitým krokom. Typickým problémom multibit metód je plytvanie alokovaným priestorom v riedkych častiach stromu. Vďaka svojim nesporným výhodám a jednoduchej hardwarovej implementácii sa stal tento algoritmus veľmi populárny.

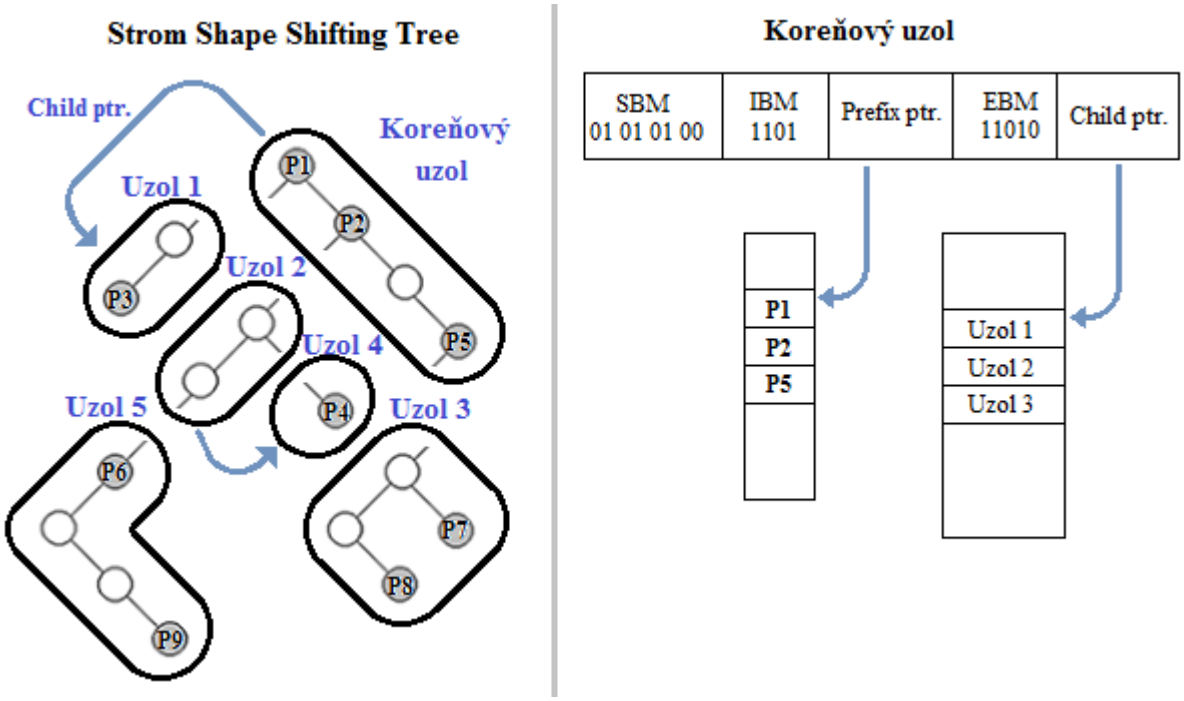
3.6 Shape Shifting Trie

Predchádzajúci algoritmus stojí na jednoduchých a zároveň efektívnych princípoch. Implementácia je priamočiara a pritom ponúka dobré výsledky. Má však svoje nedostatky, ktoré sa snaží odstrániť algoritmus Shape Shifting Trie (SST) [19]. Algoritmus Tree Bitmap pracuje s pevným počtom bitov v každom kroku a pokrýva vždy n hladín binárneho stromu. Toto je efektívne v prípade, že je binárny strom plne vyvážený. Takáto situácia však nenastáva často a priestor alokovaný pre uzol zostáva nevyužitý. Práve tento problém odstraňuje algoritmus Shape Shifting Trie, ktorý je v podstate zobecnením algoritmu Tree Bitmap.

Základné myšlienky, ktoré sa ukázali ako výhodné, zostali zachované. Tými sú zaznamenávanie potrebných informácií vo forme bitmapy a ukladanie prefixov/následníkov do polí bezprostredne za sebou. Interná a externá bitmapa, jej tvorba a práca s ňou zostali v podstate bez zmeny. Algoritmus je možné parametrizovať hodnotou K. Tá však v tomto prípade neznamená počet bitov spracovaných v jednom takte, ale maximálny počet uzlov Trie, ktoré môže zapuzdrovať jeden SST uzol. Bitmapy majú potom veľkosť K (interná bitmapa) a K+1 (externá bitmapa).

Podstatným rozdielom je pribudnutie tretej bitmapy – tvarovej (shape bitmap SBM). Tá umožňuje vytvárať uzly s rôznymi tvarmi, vďaka čomu nedochádza k plytvaniu pamäti v uzloch. Pri jej tvorbe najskôr rozšírime SST uzol o fiktívne uzly. Ak uzol Trie nemá v rámci SST uzla žiadneho následníka, dostane 2 fiktívne uzly. Ak má jedného následníka, ktorý patrí do nášho SST uzla, dostane 1 fiktívny uzol. Inak žiadny. Potom prechádzame originálne uzly v breadth-first poradí a zapisujeme hodnotu 1 pre skutočného potomka alebo hodnotu 0 pre fiktívneho potomka. Tak vznikne tvarová bitmapa o maximálnej veľkosti 2K bitov. Ukážka vytvoreného stromu algoritmom Shape Shifting Trie je na obr. 3.6.

Pri konštrukcii sa zameriame buď na minimalizáciu výšky výsledného stromu a tým pádom nižší počet prístupov do pamäte pri vyhľadávaní, alebo na minimalizáciu celkového počtu SST uzlov. Nie je však možné splniť obe kritériá zároveň. Tvorba stromu je značne komplikovaná a je tiež extrémne náročná na výpočtový výkon. Samotné vyhľadávanie prebieha rýchlo, avšak nejedná sa o triviálnu úlohu z dôvodu zvýšených nárokov na spracovanie uzlu.



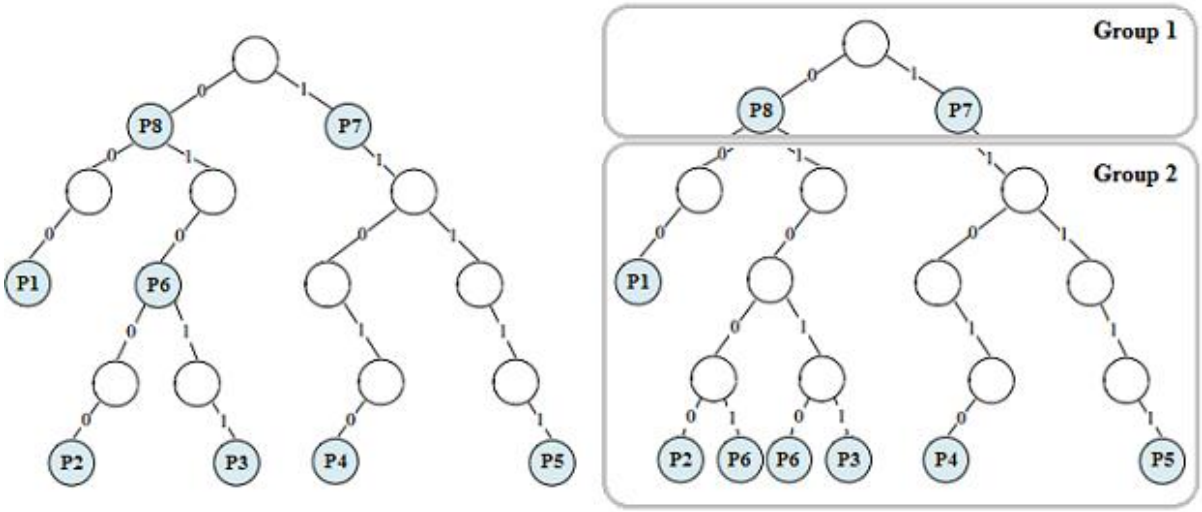
Obrázok 3.6: Dátové štruktúry algoritmu Shape Shifting Trie (K=4)

Vidíme, že algoritmus veľmi dobre zvláda prácu s pamäťou, pretože sa prispôsobuje konkrétnej prefixovej množine. Efektívne dokáže pokryť miesta s hustým vetvením, ale tiež dlhé a nevetvené úseky. Tým navyše zmenšuje výšku stromu a teda znižuje počet prístupov do pamäte a urýchľuje vyhľadanie. Táto metóda by sa na prvý pohľad mohla zdať ako ideálna, ktorá by spĺňala aj požiadavky pre nasadenie s IPv6. Kameňom úrazu sa stáva jej prehnaná komplikovanosť. V každom kroku sa spracováva rôzny počet bitov, uzly majú premenlivú veľkosť a tiež dopredu neznámy tvar. To prináša príliš dynamickú štruktúru a postup vyžadujúci náročné spracovanie každého uzlu. To všetko, ako aj ďalšie faktory, bránia hardwarovej implementácii tejto metódy.

3.7 Prefix Partitioning

Odlišný prístup pre operáciu LPM bol navrhnutý a predstavený v [9], ktorý v súčasnosti patrí medzi najefektívnejšie LPM algoritmy vhodné tiež pre nasadenie v protokole IPv6. Zameriava sa predovšetkým na minimalizáciu celkových pamäťových nárokov a optimalizáciu riedko zaplnených častí stromu.

Ako základ slúži v prvom kroku jednobitová štruktúra Trie, na ktorú sa aplikuje algoritmus tzv. „prefix partitioning“ (označovaný pod skratkou DPP). Strom Trie sa horizontálne rozdelí do *k* samostatných skupín. Na prefixy v každej skupine sa potom použije metóda „leaf pushing“, ktorá bola popísaná pri algoritme CPE. Rozdelenie do skupín sa nevykonáva s pevným krokom, ale hľadá sa také riešenie, aby sa metódou leaf pushing generoval čo najmenší počet nových prefixov. Výsledkom tejto časti je teda *k* skupín, kde každá skupina obsahuje disjunktné prefixy. To znamená, že žiadne dva prefixy sa v rámci skupiny neprekrývajú (P1 nesmie byť prefixom P2). Jednoduchá ukážka rozdeľovania pre k=2 je znázornená na obrázku 3.7.



Obrázok 3.7: Ukážka rozdeľovania prefixov do skupín pre k=2

Každá vzniknutá skupina sa následne bude spracovávať samostatne, nezávisle na ostatných. To je možné s výhodou využiť hlavne v HW implementácii, kde vďaka paralelizmu dochádza k výrazne vyššej priepustnosti. Prefixy v každej skupine sa transformujú do podoby binárneho vyhľadávacieho stromu – ľavý podstrom obsahuje hodnoty menšie ako aktuálny uzol, pravý uzol zase väčšie hodnoty. Hodnotou sa v tomto prípade myslí hodnota platného prefixu daného uzlu. Vyhľadávanie LPM potom predstavuje jednoduchý binárny vyhľadávací algoritmus na základe odpovedajúcej časti IP adresy. Výsledkom je najdlhší nájdený prefix spomedzi všetkých *k* skupín.

Výhodou použitia binárneho vyhľadávacieho stromu je priamočiary postup, na ktorý existujú optimalizované algoritmy. Nevýhodou je však nemožnosť vykonávať inkrementálne aktualizácie prefixovej množiny. Alternatívou je preto možnosť využiť tzv. 2-3 strom, ktorý tieto aktualizácie umožňuje, ale má vyššie pamäťové požiadavky. Rozdiel oproti binárnemu stromu je ten, že 2-3 strom môže obsahovať navyše uzly, ktoré majú 2 hodnoty a 3 následníkov. Jeden následník obsahuje hodnoty menšie ako prvá hodnota aktuálneho uzlu. Druhý obsahuje hodnoty, ktoré sa nachádzajú v intervale medzi prvou a druhou hodnotou aktuálneho uzlu. A posledný následník obsahuje hodnoty vyššie ako druhá hodnota uzlu.

Výhodou tohto algoritmu je dobrá pamäťová efektivita, ktorá predstavuje približne 1B pamäte potrebný na uloženie 1B prefixu, rovnako pre IPv4 a IPv6. Z dôvodu potreby disjunktných prefixov je však pamäťová zložitosť lineárna s počtom prefixov. To je viac ako u algoritmov využívajúcich myšlienku Trie, kde veľká časť prefixov obsahuje spoločné zdieľané cesty a teda sú uložené len jedenkrát. Ďalšou nevýhodou sú obrovské režijné nároky na fázu predspracovania a vytvorenie vyhľadávacieho stromu.

3.8 Porovnanie algoritmov

Algoritmy popísané v predchádzajúcich častiach práce majú viaceré podobné črty a vlastnosti. Všetky z nich stoja na využití stromovej dátovej štruktúry. Pritom je však každý z nich špecifický a od ostatných niečím odlišný. Táto podkapitola podáva zhrnutie a porovnanie uvedených LPM algoritmov, krátko diskutuje ich možnosti a významné prednosti, ktorými vynikajú.

Stručné porovnanie jednotlivých algoritmov je uvedené v tabuľke 3.1. Požité symboly značia: W – maximálna dĺžka prefixu, N – počet prefixov v množine, k – veľkosť kroku u multibit metód. Asymptotické odhady časovej a priestorovej zložitosti udávajú horný odhad v prípade najhoršej možnej situácie. Vidíme, že rýchlosť u takmer všetkých algoritmov je lineárna s dĺžkou prefixu W. Hodnota W je v najhoršom prípade 128 (pre IPv6) a môžeme ju teda považovať za konštantu. Preto je tiež rýchlosť konštantná. Výnimku tvorí DPP, ktorého rýchlosť nezávisí na dĺžke prefixu, ale na celkovom počte prefixov vo vstupnej množine. S narastajúcou prefixovou množinou sa logaritmicky zhoršuje jeho rýchlosť. Reálna rýchlosť algoritmu však nezáleží len na počte potrebných krokov, ale aj na ďalších faktoroch. Jeden krok výpočtu trvá rozdielnu dobu u každého algoritmu a odvíja sa napr. od zložitosti uzlu a náročnosti jeho spracovania. Doba spracovania jedného uzlu Tree Bitmap je napr. podstatne kratšia ako uzlu typu Shape Shifting Trie. Pre skutočné nasadenie algoritmov je v konečnom dôsledku vždy potrebná HW implementácia. Tá dovoľuje využiť akceleračné techniky ako je napr. zreťazené spracovanie a uvedené teoretické odhady rýchlosti potom strácajú svoju váhu.

Názov	Výhody	Nevýhody	Rýchlosť	Pamäť
Trie	- Jednoduchá konštrukcia - Nenáročná implementácia - Rýchla modifikácia	- Vysoký počet prístupov do pamäte - Žiadna kompresia stromu	*O(W)*	*O(NW)*
CPE	- Triviálna multibit metóda	- Vysoké pamäťové nároky - Nevhodná pre reálne použitie	*O(W / k)*	*O(2^k N W / k)*
Lulea	- Efektívna kompresia dát - Vhodná aj na uloženie v cache pamäti	- Nepodporuje aktualizácie prefixovej množiny		
LC-Trie	- Efektívna správa pamäte - Dobre pokrýva husté aj riedke časti stromu	- Dynamickosť - Nevhodné pre HW - Náročné aktualizácie		
TBM	- HW implementácia - Rýchle aktualizácie - Nízky počet ukazateľov	- Plytvanie priestorom v riedkych častiach stromu		
SST	- Prispôsobenie sa prefixovej množine - Nízke pamäťové nároky - Vysoká rýchlosť	- Premenlivé tvary uzlov - Neexistuje HW realizácia - Náročné aktualizácie		
DPP	- Pamäťová efektivita - Vhodný pre IPv6	- Vysoká réžia pre-processingu	*O(log N)*	*O(NW)*

Tabuľka 3.1: Porovnanie LPM algoritmov

Algoritmus Trie je vo svojej podstate veľmi jednoduchý a ľahko použiteľný. Implementácia dátovej štruktúry a vyhľadávacích operácií využíva základné princípy binárneho stromu. Z toho vyplýva tiež rýchle a nenáročné pridávanie a odoberanie prefixov. Jeho hlavnou nevýhodou je však príliš veľký počet prístupov do pamäte. Pre IPv6 adresy by to bolo až 128 prístupov na jedno vyhľadanie, čo je absolútne neprístupné. Tento strom nie je žiadnym spôsobom komprimovaný, čo sa odzrkadľuje tiež v pomerne vysokých pamäťových nárokoch. Na vloženie jedného prefixu

potrebujeme v najhoršom prípade pridať celú cestu (W uzlov). Pri celkovom počte N prefixov nás to vedie na priestorovú zložitosť *O(NW)*.

Nasledujúce algoritmy patria do kategórie multibit metód a rôznym spôsobom sa snažia optimalizovať základnú myšlienku Trie a tak odstrániť jej nedostatky. Asymptotický odhad zložitosti je však pre všetky multibit algoritmy rovnaký. Pri spracovaní k bitov adresy v jednom kroku sa počet krokov potrebných na vyhľadanie logicky zníži *k*-krát, teda *O(W / k)*. Pri pridaní jedného prefixu do stromu musíme potom v najhoršom prípade vytvoriť cestu dĺžky *W / k*. Každý uzol tejto cesty by mal 2^k interných položiek (vhodným spôsobom kódované pôvodné uzly Trie). Z toho dostávame priestorovú zložitosť multibit algoritmov - $O(2^k * N * W / k)$. Výnimkou je opäť DPP postavený na metóde binárneho vyhľadávacieho stromu. Každý jeho uzol obsahuje celú hodnotu svojho prefixu, a teda pri počte N prefixov dostávame priestorovú zložitosť *O(NW)*.

Algoritmus CPE jednoduchým spôsobom ukazuje základnú ideu multibit LPM metód. Vytvorené uzly stromu sú jednoduché na spracovanie a operácia vyhľadávania je tiež priamočiara. Metóda však nedokázala naplno využiť potenciál a vykazuje zásadné nedostatky. Hlavným z nich je vysoký počet ukazateľov a neefektívne hospodárenie s priestorom. Potrebnú pamäť je možné znížiť takmer na polovicu s optimalizáciou „leaf pushing“. Celkové nároky napriek tomu zostávajú neprijateľne vysoké a algoritmus nie je vhodný na reálne nasadenie.

Oproti tomu Lulea poskytuje veľmi účinnú kompresiu s využitím bitmapy na zakódovanie uzlu a odstránenie redundatných informácií. Vďaka tomu dokáže aj obrovské smerovacie tabuľky reprezentovať v kompaktnej forme. Všetky potrebné štruktúry je potom možné uložiť v rýchlych cache pamätiach, čo zaručuje výrazný výkonnostný nárast. Hlavnou nevýhodou je nepodporovanie inkrementálnej aktualizácie. Preto je nutné kompletne rekonštruovať celú tabuľku a vybudovať nový vyhľadávací strom.

Častým nedostatkom multibit algoritmov je, že dochádza k plytvaniu alokovaným priestorom v riedkych častiach stromu. Sú to miesta, kde nedochádza k častému vetveniu a prevládajú dlhé nevetvené úseky. LPM metóda LC-Trie kombinuje viacero prístupov a do istej miery odstraňuje tento jav. Zároveň pracuje efektívne aj v miestach hustého zaplnenia – miesta, kde sa strom často vetví a približuje sa k plne vyváženému stromu. Algoritmus teda veľmi efektívne pracuje s pamäťou a poskytuje vysokú rýchlosť vyhľadania. Tento návrh je však príliš dynamický, komplikovaný a vo výsledku nevhodný pre hardwarovú implementáciu. Podobne ako pri algoritme Lulea, sú aktualizácie mimoriadne zložité.

Algoritmus známy pod názvom Tree Bitmap je jeden z najrevolučnejších a najčastejšie používaných LPM algoritmov, ktorý mnohokrát potvrdil svoje výborné vlastnosti. Medzi jeho hlavné kvality patrí jednoduchá hardwarová implementácia a vysokorýchlostné aktualizácie. Podobne ako algoritmus Lulea, používa ku kompresii dát bitmapy a vyniká vysokou rýchlosťou. Pomocou uloženia nasledovníkov/prefixov do poľa bezprostredne za sebou dosiahneme výrazne menší počet potrebných ukazateľov. Jeho pamäťové nároky sú ale do značnej miery závislé na charakteristike vstupnej množiny. Mnohokrát dochádza k neefektívnemu využívaniu alokovaného priestoru a k plytvaniu pamäťou v oblasti málo zaplnených častí stromu. Tento problém sa ešte prehlbuje pri IPv6 a priestorová náročnosť prudko rastie.

Posledný multibit algoritmus – Shape Shifting Trie zachováva mnohé výhody metódy Tree Bitmap, z ktorej vychádza. Navyše odstraňuje plytvanie pamäte v málo zaplnených častiach stromu. Veľmi dobre sa dokáže prispôsobiť vstupnej množine a znižuje potrebnú pamäť na minimum. Popritom sa tiež znižuje výška vytvoreného stromu – a teda počet potrebných krokov. Rýchlosť vyhľadania by potom mala byť znateľne vyššia. Premenlivý počet spracovávaných bitov a tvary uzlov však opäť vedú k príliš dynamickej štruktúre a teda k nevhodnosti pre hardwarovú realizáciu.

Na záver je uvedený algoritmus Prefix Partitioning, ktorý oproti predchádzajúcim využíva Trie len v úvodnej fáze a následne stojí na myšlienke jednoduchého binárneho vyhľadávacieho stromu. Dosahuje dobrú pamäťovú efektivitu zhodnú pre obe verzie protokolu a je teda vhodný na univerzálne použitie. Vyhľadávanie je síce rýchle, ale vyžaduje si obrovské režijné nároky vo fáze predspracovania dát. Transformovať vstupnú prefixovú množinu do optimálnej podoby je výpočtovo značne náročná úloha. Pamäťová zložitosť navyše so zvyšujúcim sa počtom prefixov rastie rýchlejšie ako u predchádzajúcich algoritmov.

4 Analýza súčasného stavu

V predchádzajúcich kapitolách som prezentoval niekoľko známych LPM algoritmov, vysvetlil ich princípy, dátové štruktúry, vlastnosti a stručne uviedol ich teoretické výkonnostné zhodnotenie z pohľadu časovej a priestorovej zložitosti. To nám poskytlo základnú predstavu o rôznych prístupoch k problematike, niektoré optimalizačné návrhy a prednosti algoritmov, ktoré by bolo možné ďalej rozvíjať a zdokonaľovať.

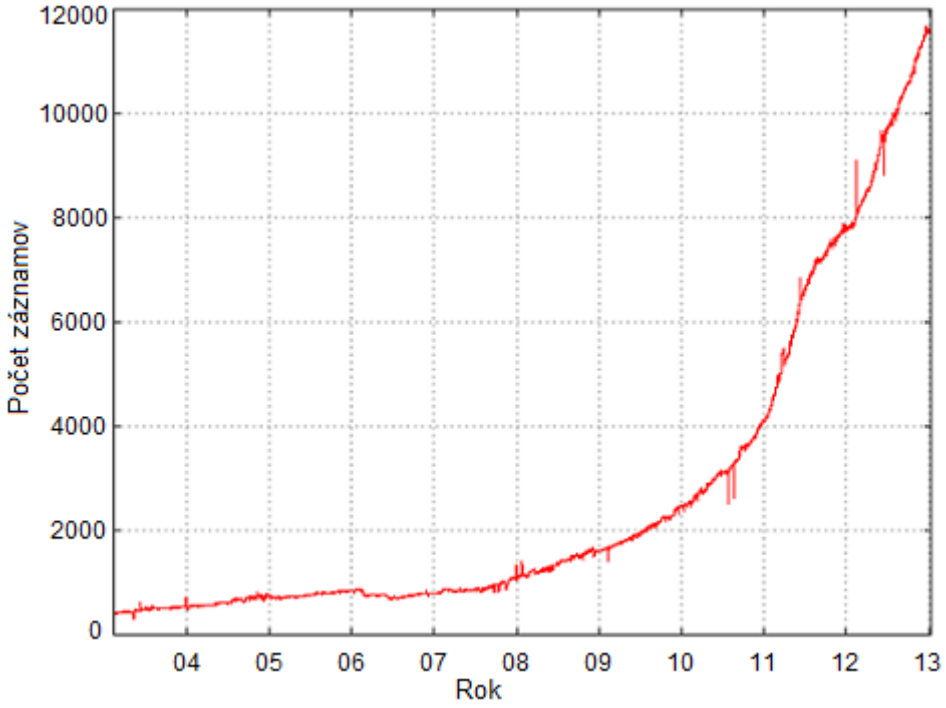
Doposiaľ však boli diskutované prevažne teoretické znalosti. Táto kapitola sa zameriava na reálne výsledky a jej hlavným zámerom je komplexná analýza uvedených algoritmov z pohľadu rýchlosti vyhľadávania, pamäťovej náročnosti a celkovej výkonnosti v kontexte protokolu IPv6. K účelom tohto testovania bol použitý nástroj Netbench [14]. Tento nástroj vznikol v rámci výskumnej skupiny ANT@FIT a je implementovaný v jazyku Python. Jedná sa o softwarovú simuláciu algoritmov, ktoré však zohľadňujú všetky aspekty skutočnej hardwarovej realizácie. Hodnoty získané týmto spôsobom môžeme považovať za dostatočne presné, ktoré reflektujú reálne výsledky.

Podstatnou časťou kapitoly je analýza prefixových množín protokolu IPv6, ktoré sa výrazne odlišujú od predchádzajúcej verzie. Na základe tejto analýzy je potom možné identifikovať typické rysy množín a vybrať algoritmy s najväčším potenciálom na optimalizácie s cieľom ponúknuť vysokú rýchlosť, nízku pamäťovú spotrebu a dosiahnuť priepustnosť 100Gbps. Získané informácie je nevyhnutné dôkladne spracovať a využiť ich v nasledujúcom návrhu.

4.1 Dostupné prefixové množiny

V súčasnosti, kedy protokol IPv4 dosiahol vrchol a narazil na svoje limitné obmedzenia, nie sú smerovacie tabuľky tohto protokolu pre experimenty príliš prínosné. Môžeme predpokladať, že sa dostali na svoju maximálnu teoretickú veľkosť. Zmeny a dynamika u nich samozrejme naďalej zostáva, ako tiež zmeny v rozložení prefixov a podobne. Ale ich veľkosť už nemôže príliš narásť, pretože voľné IPv4 adresy boli vyčerpané. Existujúce LPM algoritmy pomerne dobre zvládajú tabuľky týchto adries a nepredstavujú preto motiváciu na ich zdokonaľovanie. Z toho dôvodu sa v analýze zameriavam výhradne na protokol IPv6, kde je situácia presne opačná.

Ako sa predchádzajúca verzia protokolu blížila ku svojmu koncu, rástla dôležitosť prechodu na protokol IPv6. Tento trend môžeme pozorovať na obrázku 4.1, kde vidíme prudký nárast IPv6 adries v posledných rokoch. Obrázok ukazuje jednu smerovaciu tabuľku typu BGP (Border Gateway Protocol), ktoré sa využívajú na smerovanie medzi rozsiahlymi autonómnymi systémami. Sú to jedny z najväčších voľne dostupných smerovacích tabuliek, ktoré je možné využiť na testovanie a experimenty [1]. Ďalším dostupným zdrojom je služba SixXS [18], ktorá sprístupňuje niektoré smerovacie tabuľky získané od regionálnych registrátorov RIR. Tieto množiny sa pohybujú rádovo v stovkách až tisíckach záznamov. Ekvivalentné tabuľky IPv4 však obsahujú až stovky tisíc záznamov.



Obrázok 4.1: Vývoj veľkosti BGP tabuliek pre protokol IPv6

Z vyššie popísaných faktov vyplýva, že zdroje poskytujúce prefixové množiny k účelom analýzy a experimentom sú značne obmedzené. Navyše ich maximálna veľkosť je relatívne malá a nedokážu dostatočne reflektovať budúci rast a vývoj IPv6 sietí. Je možné využívať tiež množiny, ktoré sú náhodne generované a teda vieme dosiahnuť ľubovoľnú veľkosť. Ako však ukazuje analýza IPv4 tabuliek, histogram dĺžky prefixov neodpovedá náhodnému rozloženiu. Takto náhodne syntetizované množiny preto nespĺňajú požiadavky a neposkytujú dostatočne relevantné dáta. Návrh generovania tabuliek popísaný v [23] berie v úvahu viaceré faktory – alokačná politika, smerovacie praktiky a vývoj internetu. To sú kľúčové oblasti, ktoré vplývajú na formovanie smerovacích tabuliek. S ohľadom na tieto prvky je potom možné z existujúcich IPv4 tabuliek vygenerovať odpovedajúce IPv6 množiny o takmer zhodnej veľkosti. Takto získané tabuľky predstavujú v súčasnosti najlepší zdroj dát s cieľom otestovať predpokladané odhady tabuliek v budúcnosti. V mojej práci využívam jednak dostupné reálne zdroje a tiež množiny syntetizované týmto spôsobom. V rámci analýzy však dávam prednosť reálnym tabuľkám a syntetizované alternatívy používam až v prípade experimentov a testovania.

4.2 Analýza LPM algoritmov

V tejto časti sa bližšie zameriam na analýzu jednotlivých algoritmov z pohľadu ich reálneho nasadenia v kontexte protokolu IPv6. Ak uvažujeme o skutočnom použití LPM algoritmu v praxi, jedinou možnosťou je v súčasnosti jeho hardwarová implementácia. Iným spôsobom nie sme schopní dosiahnuť požadované výkonnostné parametre, ktoré predstavujú vyše 160 miliónov vyhľadaní za sekundu pre dosiahnutie priepustnosti na úrovni 100 Gbps. To znamená, že každé vyhľadanie musí byť uskutočnené v čase približne 6ns.

Počítačové čipy sú dnes rýchle a poskytujú pomerne vysoký výkon, ktorý je dostatočný na spracovanie paketov v uvedenej rýchlosti. Úzkym hrdlom sa však zvyčajne stáva prístup do pamäte. Narastajúce veľkosti smerovacích tabuliek spolu s protokolom IPv6 spôsobujú čoraz vyššie pamäťové nároky. LPM algoritmy teda pre svoju prácu zvyčajne vyžadujú použitie externej pamäte, v ktorej sú uložené dátové štruktúry potrebné k vyhľadávaniu. Externá pamäť má však vysokú latenciu, kvôli ktorej sa znižuje výkon a priepustnosť celého riešenia.

Hlavným sledovaným kritériom, na ktoré sa sústredím, je preto priestorová zložitosť algoritmov. Minimalizovaním pamäťových nárokov totiž môžeme docieliť takú úroveň kompresie, že bude možné všetky potrebné dátové štruktúry uložiť do internej pamäte zariadenia, ktorá je limitovaná svojou malou kapacitou. Interná pamäť má jednak nižšiu spotrebu, ale hlavne omnoho nižšiu prístupovú dobu. To prináša ohromný nárast výkonnosti a teda tiež rýchlosti vyhľadávania. Priepustnosť algoritmu je možné ďalej zvyšovať využitím typických hardwarových techník, akými je napríklad paralelizmus alebo zreťazené spracovanie. Dostatočná dĺžka zreťazenej linky nám potom dovoľuje dostať sa na teoretické maximum, kedy v každom takte získavame výsledok vyhľadania pre jeden paket.

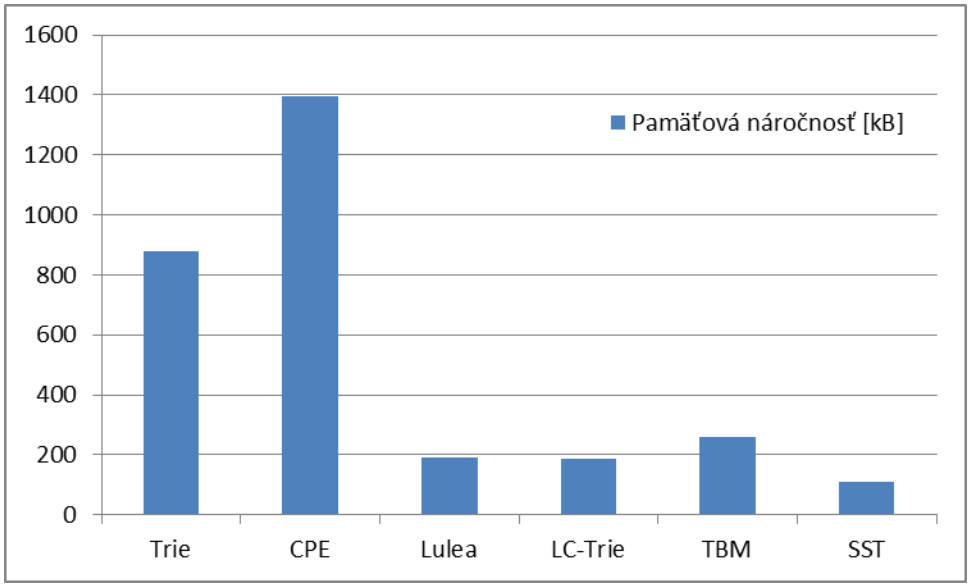
V tabuľke 4.1 sú zhrnuté výsledky jednotlivých algoritmov z pohľadu ich pamäťovej spotreby. Tieto výsledky sú tiež zobrazené v podobe grafu na obrázku 4.2. Experimenty boli vykonané nad reálnymi IPv6 množinami s veľkosťou približne 10 000 prefixov. Pri mulitbit metódach bol použitý krok n=4 a bolo uvažované použitie ukazateľov veľkosti 32 bitov. Do experimentov nebol zahrnutý algoritmus Prefix Partitioning, pretože jeho implementácia nie je dostupná v knižnici Netbench. K dispozícii sú len výsledky algoritmu [9], kde však bola použitá odlišná metodika. Tento algoritmus preto nie je možné zaradiť do porovnania výsledkov v rámci analýzy.

Algoritmus	Počet uzlov	Pamäť [kB]	Výška stromu
Trie	75057	879,57	128
CPE	22298	1393,62	32
Lulea	22298	191,26	32
LC-Trie	16041	187,98	20
TBM	22298	258,58	32
SST	7339	111,98	13

Tabuľka 4.1: Porovnanie pamäťovej náročnosti algoritmov

Algoritmus Trie kóduje prefixovú množinu priamo vo svojej konštrukcii v podobe jednoduchého binárneho stromu a nevyužíva žiadnu kompresiu. Nie je síce vhodný na reálne nasadenie, ale z pohľadu analýzy nám môže dobre poslúžiť ako referenčná hodnota pri porovnávaní efektivity ostatných algoritmov. Keďže sa jedná o adresy IPv6, je výška stromu unibit Trie 128, čo znamená až 128 prístupov do pamäte na uskutočnenie jedného vyhľadania.

Pri pohľade na počet uzlov a výšku stromu pri algoritmoch CPE, Lulea a Tree Bitmap vidíme, že sa ich hodnoty zhodujú. Je to spôsobené tým, že sú všetky založené na rovnakom princípe pokrývania stromu Trie. Pri nastavenom parametri stride 4, spracovávajú v každom kroku 4 bity adresy a teda je strom 4-krát nižší ako pri unibit Trie (128/4 = 32). Každý vytvorený multibit uzol potom môže pokrývať až $2^4 - 1 = 15$ unibit uzlov. Preto je namapovanie mulitbit uzlov totožné. Pamäťové nároky sa však líšia podľa reprezentácie uzlu.



Obrázok 4.2: Porovnanie pamäťovej náročnosti algoritmov

Algoritmu CPE vykazuje extrémnu pamäťovú náročnosť, ktorá dokonca presahuje jednoduchú Trie. To je spôsobené veľkým počtom potrebných ukazateľov. Pri stride 4 potrebuje každý uzol až 2^4 odkazov, čím sa dostávame na 512 bitov potrebných pre každý uzol. Tento prístup sa ukazuje ako absolútne neefektívny a neprijateľný. Algoritmus neponúka žiadne významné prínosy, ktoré by sme mohli využiť, a preto nemá zmysel mu venovať veľkú pozornosť.

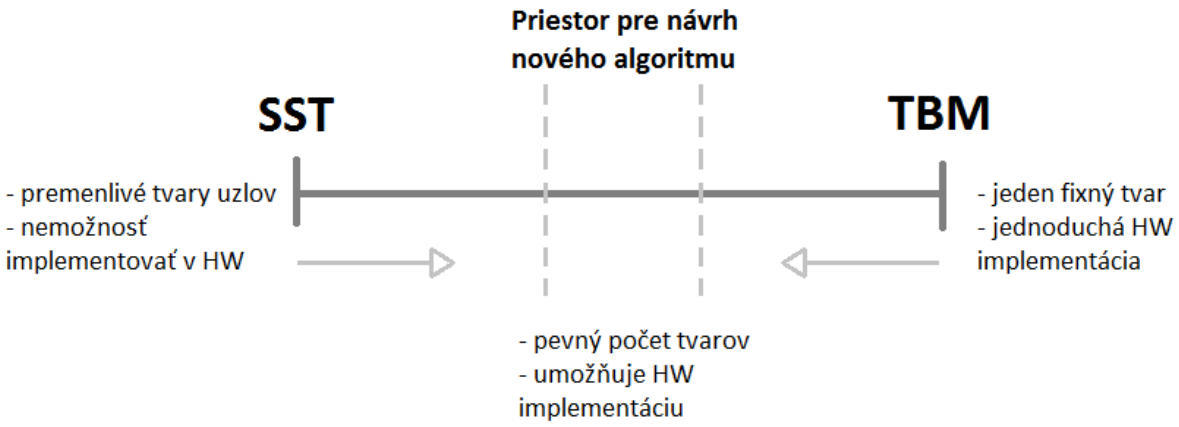
Nasledujúce 3 metódy (Lulea, LC-Trie a Tree Bitmap) vykazujú porovnateľné výsledky. Lulea a Tree Bitmap sú založené na podobnej myšlienke – využiť bitmapy na redukovanie veľkosti uzla. V zásade sú však výrazne odlišné. Tree Bitmap nepoužíva „leaf pushing“, a preto poskytuje vysokorýchlostné aktualizácie, ktoré sú pri Lulea nemožné. Ďalšou výhodou Tree Bitmap je kompletné spracovanie uzlu v 1 pamäťovom prístupe (oproti 3 prístupom pri Lulea). Tree Bitmap je veľmi dobre optimalizovaný pre hardwarové nasadenie, kde ponúka jednoduché, ale robustné riešenie. Hoci naše výsledky ukazujú v porovnaní s metódou Lulea mierne vyššie hodnoty, vďaka uvedeným prednostiam má algoritmus Tree Bitmap lepšiu perspektívu a uplatnenie.

Metóda LC-Trie dosahuje pamäťovú spotrebu takmer identickú s algoritmom Lulea. Ako vidíme, výška stromu je však nižšia, čo predstavuje lepšiu teoretickú rýchlosť. Túto výhodu získava vďaka tomu, že dokáže efektívne preskočiť dlhé nevetvené úseky stromu. Táto vlastnosť sa zdá byť nádejným a zaujímavým prvkom práve pri protokole IPv6. Charakter uzlov a spracovanie vstupnej informácie však bráni hardwarovej realizácii, čo je veľmi výrazný nedostatok algoritmu.

Jednoznačným víťazom uvedených experimentov sa stáva metóda Shape Shifting Trie. Minimalizuje nielen potrebný pamäťový priestor, ale tiež výšku výsledného stromu. Obe hodnoty sú výrazne nižšie v porovnaní s ostatnými metódami. Dosahuje to vďaka tomu, že uzly tohto stromu sa svojím tvarom dokonale prispôsobujú konkrétnej prefixovej množine. Jediným, avšak zásadným nedostatkom je nemožnosť hardwarovej implementácie.

Algoritmus Shape Shifting Trie však vychádza z metódy Tree Bitmap, ktorá naopak vykazuje jednoduchú HW implementáciu. Dá sa povedať, že Tree Bitmap je len špecifický prípad algoritmu Shape Shifting Trie, kedy je povolený jeden fixný tvar používaných uzlov. Vlastnosti týchto dvoch príbuzných metód dávajú priestor pre návrh nového algoritmu, ktorý bude spájať jednoduchú hardwarovú realizáciu, minimalizáciu pamäťových nárokov a zároveň dostatočne vysokú

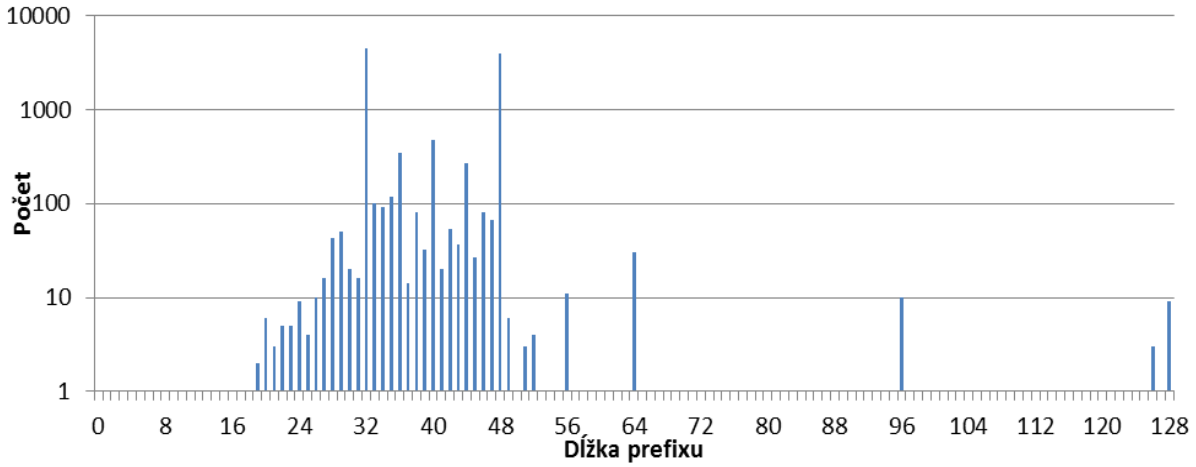
priepustnosť a rýchlosť vyhľadávania. K dosiahnutiu uvedeného cieľa je nutné obmedziť povolené tvary SST len na niekoľko základných typov. Respektíve pridať algoritmu TBM niekoľko nových tvarových uzlov. Tento myšlienkový postup s identifikáciou priestoru pre nový algoritmus je ilustrovaný na obrázku 4.3. Aby sme však vedeli, ktoré tvary by boli ideálne, je potrebné analyzovať typické prefixové množiny protokolu IPv6. Touto analýzou je možné identifikovať ich charakteristické rysy a použiť tieto informácie pri návrhu nového algoritmu.



Obrázok 4.3: Identifikovanie priestoru pre návrh nového algoritmu

4.3 Analýza IPv6 množín

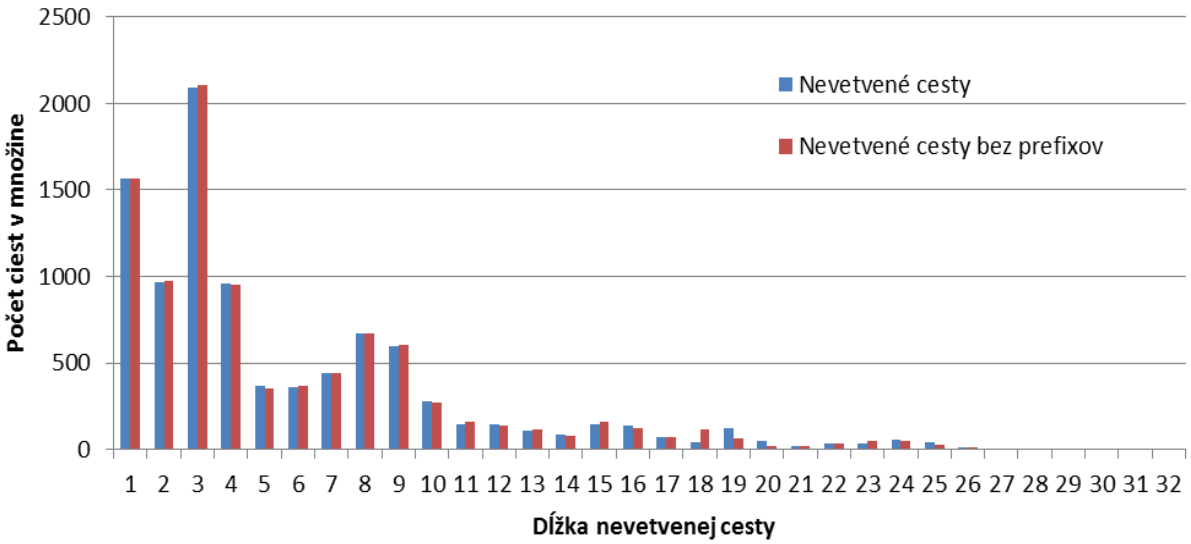
Množiny protokolu IPv6 sa výrazne odlišujú od predchádzajúcej verzie a vykazujú isté typické znaky. Hlavným rozdielom, ktorý spôsobuje najväčšie problémy je samozrejme maximálna dĺžka prefixov. Túto analýzu som vykonával na najväčších množinách, ktoré sú v súčasnosti dostupné a predstavujú situáciu pri smerovaní medzi rozsiahlymi autonómnymi systémami na úrovni RIR. Na obrázku 4.4 vidíme histogram rozloženia prefixov v množine podľa dĺžky prefixu. Graf má použitú logaritmickú mierku a vidíme obrovskú prevahu prefixov dĺžky /32 a /48. Nie je to samozrejme náhoda, ale takáto charakteristika má úzku súvislosť s alokačnou politikou popísanou v kapitole 2.4. Bloky adries tejto veľkosti sa prideľujú veľkým organizáciám a ISP. Ak by sme skúmali smerovacie tabuľky menších sietí, môžeme očakávať nárast prefixov veľkosti /56, ktoré sú typické pre domácnosti alebo malé firmy. Ďalej na histograme vidíme, že takmer všetky prefixy sú zhromaždené v prvej polovici grafu. Tento jav má opäť spojitosť s alokačnými a smerovacími praktikami. Na účely smerovania sa typicky využíva len prvých 64 bitov adresy a spodná časť sa zvyčajne použije až v koncovej lokálnej sieti pre doručenie paketu cieľovému počítaču. Preto sa prefixy takýchto dĺžok v smerovacích tabuľkách vyskytujú len zriedkavo a v malom množstve.



Obrázok 4.4: Histogram rozloženia IPv6 prefixov v množine

Ďalší znak, ktorý si určite všimneme je, že sa v množine nachádzajú pomerne veľké medzery, čo vedie na dlhé a nevetvené cesty v prefixovom strome. Dá sa predpokladať, že takéto cesty sa nachádzajú aj v nižších úrovniach stromu. Túto myšlienku však pomocou analýzy histogramu nedokážeme overiť. Dobre nám však môže poslúžiť algoritmus Shape Shifting Trie, ktorý svojím tvarom kopíruje charakter množiny. Skúmaním tvarových bitmáp som zistil, že v priemere vyše 80% uzlov má tvar nevetvenej cesty, ktoré sa ďalej odlišujú tvarom a výškou. Často sa však vyskytovali dlhé rovné reťazce so smerovaním doľava. Tento vzor sa vyskytuje zase z dôvodu charakteru IPv6 adries, kde sú časté adresy s veľkým počtom nulových bitov (napr. 1080:0:0:0:0:0:200C:417A). Ďalším typickým tvarom bol uzol obsahujúci jedno rozvetvenie. Ten sa však už vyskytoval v podstatne menšom množstve. Dlhé a nevetvené úseky sa ukazujú ako dobré potenciálne miesto na aplikovanie kompresie, a preto sa vyplatí venovať mu pozornosť.

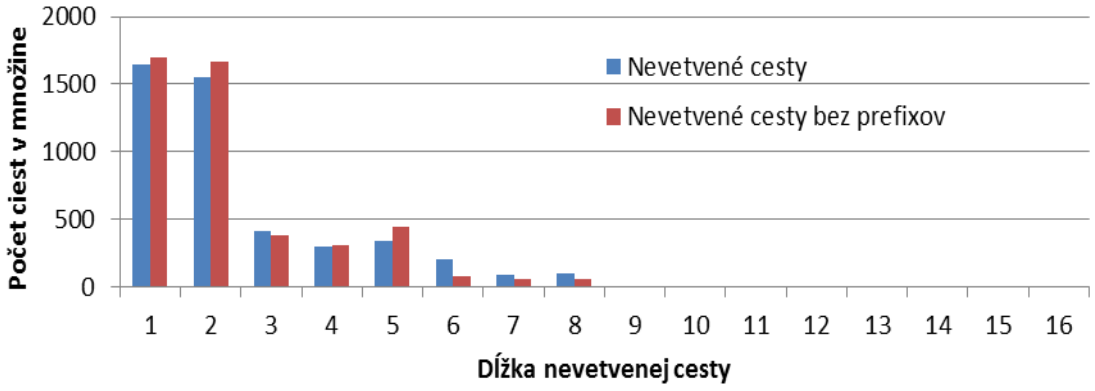
K účelom analýzy je výhodné využiť tiež jednoduchú unibit Trie. Jej strom je totiž presnou kópiou vstupnej množiny a poskytne nám dobrú predstavu o ďalších vlastnostiach. Graf na obrázku 4.5 ukazuje rozloženie nevetvených ciest v prefixovej množine. Každý stĺpec vynesený v grafe predstavuje počet výskytov nevetvenej cesty danej dĺžky. Nevetvená cesta dĺžky jedna znamená, že medzi dvoma vetviacimi uzlami sa nachádza jeden uzol, ktorý má práve jedného následníka. Dĺžka nevetvenej cesty teda odpovedá počtu uzlov, ktoré by sme teoreticky mohli preskočiť. Vidíme, že počet takýchto ciest je obrovský a s narastajúcou dĺžkou cesty sa výskyt logicky znižuje. V množinách existujú aj nevetvené cesty, ktoré majú viac ako 32 bitov. Ich počet je však zanedbateľne malý. Ešte zaujímavejšie zistenie ukazujú červené položky vyznačené v grafe. Tie značia nevetvené úseky, ktoré navyše neobsahujú vo svojich uzloch žiadny platný prefix. Prekvapivo sa tieto cesty takmer presne zhodujú s predchádzajúcimi. Všetky takéto cesty by sme mohli bez straty akejkoľvek dôležitej informácie preskočiť, napríklad spôsobom akým to vykonáva algoritmus LC-Trie. Počet takto vynechaných uzlov potom predstavuje až 73% celkového počtu uzlov. Takýto krok sľubuje značnú pamäťovú úsporu.



Obrázok 4.5: Výskyt nevetvených ciest v množine podľa ich dĺžky

V predchádzajúcej kapitole analýzy nám vzišli 2 najperspektívnejší kandidáti na možnosti ďalšej optimalizácie – Tree Bitmap a Shape Shifting Tree. Druhý z nich výborne pracuje s protokolom IPv6, ale neumožňuje hardwarovú realizáciu, čo je závažná prekážka. Priblížme si preto analýzu prefixových množín z pohľadu algoritmu Tree Bitmap a pokúsme sa odhaliť dôvody, prečo v tomto kontexte dochádza k poklesu jeho efektivity.

Obrázok 4.6 ukazuje znovu nevetvené úseky v množine, tentokrát pri aplikovaní algoritmu Tree Bitmap s parametrom n=3. Vidíme ekvivalentné výsledky s obrázkom 4.5, avšak cesty sú približne 3-krát kratšie kvôli použitému kroku 3. Nevetvená cesta sa skladá z TBM uzlov, ktoré majú vždy práve 1 následníka. Nevetvené cesty bez prefixov obsahujú len také TBM uzly, ktoré neobsahujú žiadny prefix – ich interné bitmapy sú vždy nulové. Je možné ich teda tiež bez straty preskočiť. Počet takýchto uzlov je pre dané množiny až 45%. To je jeden z hlavných dôvodov neefektivity algoritmu Tree Bitmap. Všetky uzly majú alokovaný rovnaký priestor – interná bitmapa (2^n - 1 bitov), externá bitmapa (2^n bitov) a 2 ukazatele. Veľká časť z toho je však nevyužitá a dochádza k plytvaniu, ktoré je najvýraznejšie práve pri nevetvených úsekoch stromu. A ako je vidieť, výskyt takýchto úsekov je v protokole IPv6 veľmi častý.



Obrázok 4.6: Nevetvené cesty v množine pri použití Tree Bitmap (n=3)

Na danom grafe nás môže spočiatku zaraziť situácia, že počet ciest bez prefixov (červené položky) je v niektorých prípadoch vyšší ako celkový počet nevetvených úsekov tejto dĺžky (modré položky). Takáto situácia je zreteľná napr. pri úsekoch dĺžky 1 alebo 2. Tento jav sa vyskytoval už pri Trie, ale nebol tak výrazne viditeľný. Vysvetlenie je jednoduché – do celkového počtu sa započítavajú všetky nevetvené úseky bez ohľadu na to, či obsahujú nejaké prefixy. Tak často vznikajú veľmi dlhé úseky. Ak však takáto dlhá cesta obsahuje niekde vo svojej časti aspoň jeden prefix, rozdelí sa na 2 alebo viac bezprefixových úsekov. Zvyšuje sa preto počet krátkych bezprefixových ciest a kompenzuje sa to nižším počtom dlhých úsekov.

Zaujímavý pohľad na využitie Tree Bitmapových uzlov nám podáva tabuľka 4.2. Môžeme ju interpretovať ako dvojrozmerný histogram, kde x-ová os (stĺpce tabuľky) ukazuje počet následníkov uzlu a y-ová os (riadky tabuľky) zase počet prefixov, ktoré uzol obsahuje. Na priesečníku potom nájdeme hodnotu, ktorá vyjadruje, koľko takýchto uzlov sa v strome nachádzalo. Pri pohľade na údaje si všimneme, že najvyššie hodnoty sa nachádzajú v prvom riadku tabuľky. Ten odpovedá uzlom, ktoré v sebe neobsahujú žiadny platný prefix. Slúžia teda len ako prechodné uzly, pomocou ktorých sa dostávame do nižších častí stromu. Maximálna hodnota je práve pre situáciu: 1 následník, 0 prefixov, teda spomínané nevetvené úseky bez prefixov. Ako je vidieť, významné sú tiež bezprefixové uzly s 2 následníkmi. So zvyšovaním stupňa rozvetvenia sa ale počet takýchto uzlov znižuje. Ďalšia oblasť, ktorá predstavuje potenciál pre optimalizácie sa ukazuje v prvom stĺpci tabuľky, ktorý reprezentuje listové uzly stromu. Opäť sa potvrdzuje, že sú tieto uzly veľmi málo využité, keďže obsahujú vo väčšine prípadov len 1 platný prefix. Vyššie popísané typy uzlov majú najsilnejší vplyv na pamäťové nároky, keďže sú dominantné. Zvyšné uzly tvoria zanedbateľné množstvo a nevykazujú charakteristické vlastnosti, ktoré by bolo možné využiť.

		Počet následníkov uzlu								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Počet prefixov	**0**	0	**11303**	**1666**	**812**	**538**	**184**	**145**	**131**	**249**
	1	**8965**	547	142	19	17	3	2	1	1
	2	**193**	21	14	4	3	0	1	0	0
	3	**50**	3	3	3	1	0	1	0	0
	4	**29**	3	1	1	3	1	1	0	0
	5	**0**	1	0	1	0	0	0	0	0

Tabuľka 4.2: Využitie Tree Bitmap uzlov (n=3)

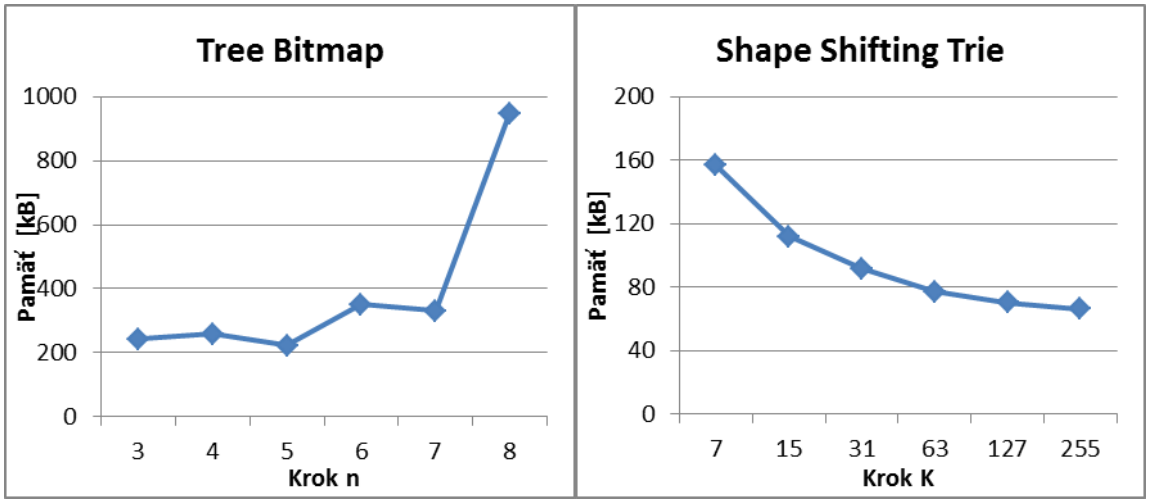
Ako už bolo viackrát potvrdené predchádzajúcimi experimentmi, obsahujú množiny IPv6 veľké množstvo úsekov s malým stupňom vetvenia, čo spôsobuje plytvanie priestorom pri algoritme Tree Bitmap. Efektivita využitia alokovaného priestoru závisí samozrejme tiež na použitom kroku. Túto situáciu zachytáva tabuľka 4.3 a porovnáva výsledky s algoritmom Shape Shifting Trie. Túto závislosť celkových pamäťových nárokov na použitom parametri názorne zobrazuje tiež graf na obrázku 4.7. Pri výpočtoch bolo uvažované použitie 32 bitových ukazateľov na adresovanie uzlov a prefixov. Z uvedených výsledkov vidíme, že pamäťové nároky pre kroky 3, 4 a 5 sú porovnateľné. So zvyšovaním kroku samozrejme klesá výška stromu, a preto sa z pohľadu vyváženia výkonnostných parametrov ukazuje ako najlepšie použitie kroku n=5. Ten poskytuje najnižšie pamäťové nároky a zároveň pomerne dobrú výšku stromu. S každým zvýšením kroku sa priemerná efektivita využitia

TBM uzlov znižuje. Teoretické maximum 100% využitia nie je možné dosiahnuť. Táto situácia by nastala jedine v prípade plne vyváženého stromu, ktorý sa v reálnych podmienkach nikdy nevyskytuje. Uvedená efektivita sa počíta ako pomer počtu unibit uzlov, ktoré Tree Bitmap pokrýva a maximálneho počtu unibit uzlov, ktoré by v ideálnom prípade dokázal pokryť. Hodnoty parametru 8 a prípadne vyššie sa už ukazujú ako nevyhovujúce, keďže veľkosť uzlu sa zvyšuje exponenciálne a ich efektívne využitie prudko klesá. Takto vysoké hodnoty nám dávajú len výhodu rýchlejšieho vyhľadania, ktoré však dokážeme docieliť aj inými technikami.

Rovnaké experimenty boli vykonané tiež s algoritmom Shape Shifting Trie. Hodnota kroku u tohto algoritmu je ale vyjadrená odlišným spôsobom. Nevyjadruje počet bitov spracovaných v jednom kroku, ale maximálny počet unibit uzlov, ktoré môže pokryť jeden SST uzol. Ekvivalentné hodnoty odpovedajúce vykonaným testom sú teda 7, 15, 31, 63, 127 a 255, pretože algoritmus Tree Bitmap s parametrom n má možnosť pokryť maximálne $(2^n - 1)$ unibit uzlov. Ako vidíme z uvedených výsledkov, efektivita využitia SST uzlov si zachováva výborné hodnoty a klesá len pomaly. Dosiahnuť využitie 100 % samozrejme zase nie je možné. Uzly sa svojím tvarom síce veľmi dobre prispôsobujú danej množine, vždy sa ale objavujú uzly, ktoré sú kratšie ako K bitov a teda priemerné využitie klesá. Algoritmus vykazuje zo zvyšujúcou hodnotou parametru znižujúce sa pamäťové nároky a tiež výšku stromu. Tieto hodnoty sú niekoľkonásobne lepšie ako ekvivalentné hodnoty Tree Bitmap. Nevýhodou však je, že je nutné v každom uzle navyše ukladať tvarovú bitmapu. Príliš vysoké hodnoty parametru potom prinášajú neúmerne veľké uzly. V prípade parametru k=255 a 32-bitových odkazoch je to až 255 + 256 + 2*255 + 32 + 32 = 1085 bitov na každý uzol. Zároveň dlhé bitmapy predstavujú úlohu náročnejšiu na spracovanie. Preto by bolo nutné vyhľadať také nastavenie parametrov, ktoré poskytuje najlepší pomer sledovaných kritérií.

Tree Bitmap					Shape Shifting Trie				
Krok n	Počet uzlov	Pamäť [kB]	Využitie uzlov	Výška	Krok K	Počet uzlov	Pamäť [kB]	Využitie uzlov	Výška
3	25063	241,69	42,78%	43	7	13795	156,60	77,73%	19
4	22298	258,58	22,44%	32	15	7339	111,98	68,18%	14
5	14352	222,49	16,87%	26	31	3973	91,66	60,94%	12
6	15078	351,54	7,90%	22	63	1990	77,00	59,87%	11
7	8505	331,18	6,95%	19	127	1004	70,22	58,86%	9
8	13488	946,72	2,18%	16	255	501	66,35	58,75%	8

Tabuľka 4.3: Vplyv použitého kroku na využitie uzlov a pamäťové nároky



Obrázok 4.7: Vplyv použitého kroku na pamäťovú náročnosť

Všetky analýzy tejto kapitoly boli vykonané na najväčších IPv6 množinách, ktoré sú v súčasnosti dostupné. Použitie generovaných množín by mohlo pôsobiť zavádzajúco a nemuselo by odrážať skutočnosť. Pre potreby analýzy je nevyhnutné reflektovať skutočný stav a skúmať relevantné zdroje, s pomocou ktorých je možné vyvodiť správne závery. Charakteristiky identifikované v tejto analýze poskytli cenné informácie, ktoré tvoria základ v procese návrhu nového algoritmu.

5 Návrh algoritmu

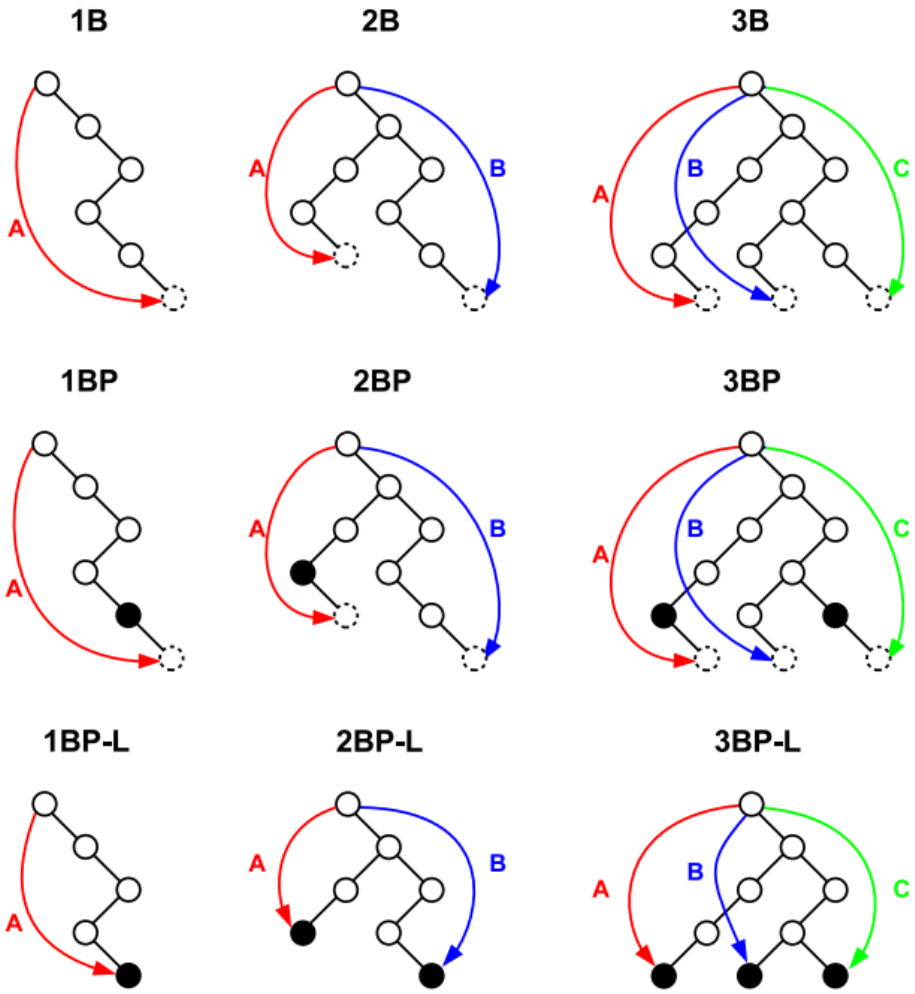
Po dôkladnej analýze známych LPM algoritmov a predovšetkým dostupných prefixových množín protokolu IPv6 máme k dispozícii množstvo cenných informácií. Boli identifikované niektoré charakteristické črty, ktoré sa vyskytujú u prefixových množín nového protokolu a predstavujú potenciál pre návrh optimalizácií.

Táto kapitola obsahuje popis návrhu nového algoritmu pre operáciu vyhľadávania najdlhšieho zhodného prefixu. Primárnym cieľom bolo navrhnúť algoritmické riešenie tejto operácie takým spôsobom, aby bola dosiahnutá maximálna úroveň pamäťovej efektivity. Vytýčenou hranicou je minimalizovanie priestorovej náročnosti do takej miery, aby bolo možné umiestniť všetky potrebné dátové štruktúry do internej pamäte FPGA. Ďalším podstatným cieľom bolo dosiahnutie priepustnosti výsledného riešenia na úrovni minimálne 100 Gbps.

5.1 Reprezentácia prefixovej množiny

Analýza uvedená v predchádzajúcej kapitole ukázala predovšetkým 2 typy uzlov, na ktoré je vhodné sa zamerať (tabuľka 4.2). Sú to listové uzly stromu, ktoré nemajú následníkov a obsahujú len odkazy na platné prefixy. Druhou výraznou skupinou sú interné prechodové uzly, ktoré neobsahujú prefixy a zvyčajne sú nevetvené alebo s nízkym stupňom vetvenia. Tieto identifikované uzly tvoria dominantnú časť celého prefixového stromu. Preto som sa snažil vytvoriť reprezentáciu, ktorá bude predstavovať efektívne zakódovanie práve týchto najčastejších situácií.

Navrhol som sadu niekoľkých typov uzlov, s ktorými následne algoritmus bude schopný pracovať. Jednoduchá grafická reprezentácia navrhnutých uzlov spolu s kódovým označením uzlov je znázornená na obr. 5.1. Uvedené obrázky značia vždy určitý úsek jednobitovej Trie, ktorý bude zapuzdrený do jedného nového multi-uzlu. Kódové názvy jednotlivých uzlov do značnej miery popisujú vlastnosti daného typu. Prvým z nich je uzol pod označením 1B (1 branch), ktorý je prechodový, neobsahuje v sebe žiadne prefixy a žiadne vetvenia. Je to teda jediná dlhá nevetvená cesta. Uzol 2B (2 branches) má zhodné vlastnosti s výnimkou, že povoľuje jedno vetvenie v rámci uzlu. A nakoniec uzol 3B, ktorý umožňuje zapuzdriť časť Trie, ktorá obsahuje až 2 vetvenia, to znamená 3 samostatné vetvy. Všetky 3 doposiaľ zmienené typy uzlov teda reflektujú jednu majoritnú skupinu identifikovaných uzlov – prechodové uzly bez prefixov s nízkym stupňom vetvenia.



Obrázok 5.1: Grafická reprezentácia novej sady uzlov

Nasledujúce 3 typy uzlov sa v názve odlišujú jedine pridaním 1 písmena „P“ na konci – teda 1BP, 2BP a 3BP. Písmeno P značí „prefix“ a znamená, že na konci každej vetvy je možné uložiť platný prefix. Tento prefix sa však môže nachádzať jedine na konci, teda v poslednom unibit uzle danej vetvy. Tieto 3 typy uzlov teda povoľujú uloženie prefixu v rámci uzlu, ale nie je však povinné. Niektoré vetvy môžu byť ukončené i bez prefixu (ako je to znázornené u 2BP a 3BP). Ostatné vlastnosti zostávajú zhodné s predchádzajúcim popisom. Využitie týchto 3 typov uzlov sa predpokladá v menšom množstve, a to v situáciách, kedy sa vyskytuje prefix vo vnútri stromu. Alebo v prípade, že jedna vetva stromu končí a nemá ďalších následníkov, zatiaľ čo ostatné vetvy pokračujú ďalej.

Posledné 3 znázornené uzly pridávajú v názve opäť len 1 písmeno – „L“, ktoré tentokrát značí „list“ (leaf). To znamená, že daný uzol je listový a nemá žiadneho následníka. Z logiky, akým sa tvorí Trie potom vyplýva, že všetky vetvy daného uzlu teda musia obsahovať prefixy. Tieto 3 typy uzlov nám zase pokrývajú druhú majoritnú identifikovanú skupinu – listové uzly stromu.

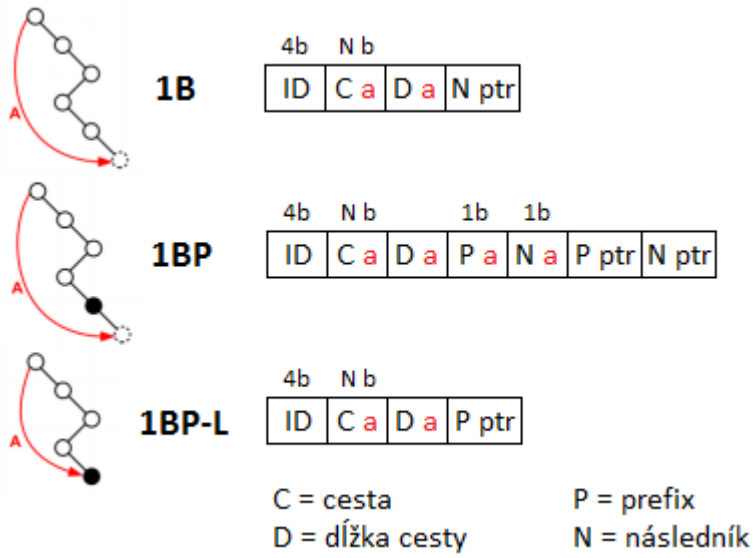
Uvedená navrhnutá sada teda obsahuje celkovo 9 nových typov uzlov, ktoré predstavujú efektívne zakódovanie úsekov stromu, ktoré boli v predchádzajúcej analýze objavené ako najvýznamnejšie a najčastejšie sa vyskytujúce (tabuľka 4.2, obr. 4.5). Občas sa však vyskytujú tiež časti, ktoré nie je možné pokryť týmito novými uzlami. Sú to predovšetkým situácie s vysokým

stupňom vetvenia alebo úseky vysokou hustotou prefixov. V takomto prípade je najefektívnejšie použiť klasický TBM uzol, ktorý je za týchto podmienok dobre využitý a nedochádza k plytvaniu pamäte. Preto som do používanej sady uzlov zaradil tiež tento typ uzlu. Dostávame teda spolu akési šablóny, s ktorými budeme ďalej pracovať a na základe určitých podmienok sa budeme rozhodovať, ktorú z nich je práve vhodné použiť.

5.1.1 Dátové štruktúry

Ako vidíme, navrhnuté uzly sa navzájom od seba vždy určitým spôsobom odlišujú, a preto tiež používané dátové štruktúry budú mierne odlišné pre každý z nich. Nie je teda možné použiť uniformnú štruktúru, ktorá by vyhovovala každému uzlu a nie je to ani žiaduce. Ak by bola použitá takáto jednotná reprezentácia pre každý uzol, dochádzalo by opäť k plytvaniu pamäťou podobne ako u algoritmu Tree Bitmap.

Jednotlivé dátové štruktúry sú však do značnej miery podobné a je zreteľná logika návrhu, ktorá je zachovaná pri všetkých uzloch. Na obrázku 5.2 vidíme potrebné dátové štruktúry pre uzol typu 1B a odvodené – teda uzly, ktoré povoľujú len jednu nevetvenú cestu.



Obrázok 5.2: Dátové štruktúry uzlov typu 1B, 1BP a 1BP-L

Na začiatku sa vždy nachádza ID uzlu, ktoré jednoznačne identifikuje o aký typ uzlu sa jedná. Túto informáciu potrebujeme načítať vždy ako prvú, aby sme dokázali správne pracovať s daným uzlom a odvíja sa od nej tiež celkový počet položiek dátovej štruktúry. Táto položka má 4 bity, a teda by sme mohli pracovať až s 16 rôznymi typmi uzlov.

Nasleduje cesta vetvy, ktorá je zadaná binárnou reprezentáciou. Je to teda postupnosť jedničiek a núl takých, aby sme sa pomocou nich dostali do uzlu následníka. Pre cestu *a* na obrázku teda odpovedá „11011“. Cesta v uzle môže mať ľubovoľný tvar a tiež variabilnú dĺžku. Vždy však existuje obmedzenie maximálnej dĺžky, ktorú nesmie presiahnuť. Tento limit sa nastavuje pomocou parametra pre každý typ uzlu zvlášť. Obecne je to teda N bitov.

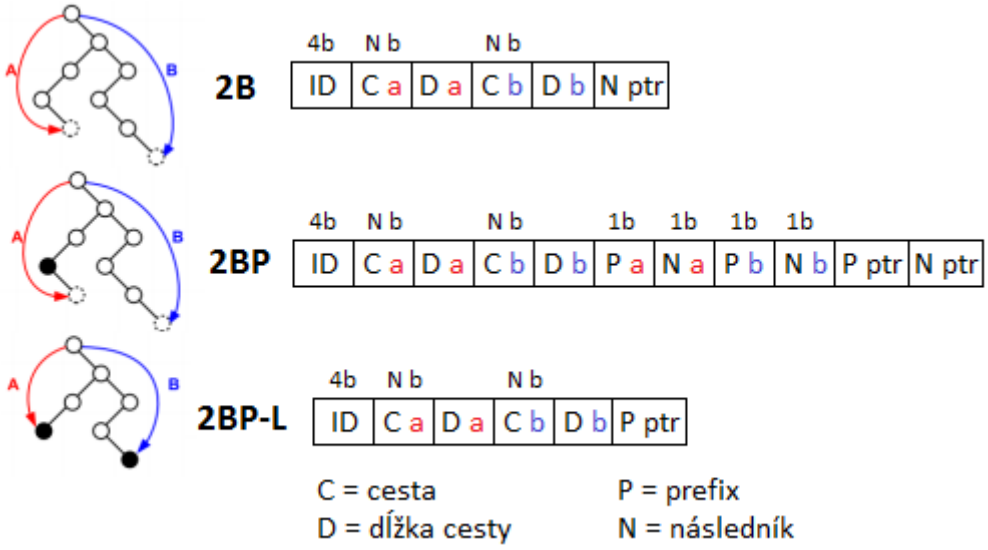
Keďže vetva v rámci uzlu môže mať variabilnú dĺžku, potrebujeme vedieť skutočnú dĺžku, ktorú vetva nadobúda. Túto informáciu nám podáva nasledujúca položka dátovej štruktúry – „dĺžka

cesty *a*“. Veľkosť tejto položky závisí od nastaveného parametra N – je to hodnota $\log_2$ N zaokrúhlená na celé čísla smerom nahor. Čiže je to minimálny počet bitov, na ktorých vieme zakódovať hodnotu N. „Cesta vetvy“ a „dĺžka vetvy“ tvoria vždy pár, ktorý jednoznačne určuje jeden konkrétny úsek stromu.

V ďalších položkách sa už uvedené uzly mierne líšia. Najvyšší počet záznamov obsahuje uzol typu 1BP, ktorý na konci vetvy môže (ale nemusí) obsahovať prefix. Nasledujúca jednobitová položka teda určuje, či sa na konci vetvy prefix nachádza (hodnota 1) alebo nie (hodnota 0). Je to teda akási jednoduchá interná bitmapa. Analogicky je uložená informácia o následníkoch, ktorá určuje, či daný uzol má/nemá následníka (1/0). Na konci štruktúry sa nachádzajú odpovedajúce ukazatele, ktoré odkazujú na platné prefixy a následníkov.

Z logiky, akou je zostavený prefixový strom Trie vyplýva, že každá vetva unibit Trie musí byť ukončená prefixom (nesmie existovať listový uzol, ktorý neobsahuje prefix). Tento poznatok môžeme využiť v uzloch typu 1B a 1BP-L. Vďaka tomu u nich nepotrebujeme uchovávať bitmapy. Uzol typu 1B nemá povolené žiadne prefixy, a preto logicky musí obsahovať následníka. Podobne uzol typu 1BP-L nemá povolených žiadnych následníkov, a preto rozhodne musí obsahovať prefix. Vďaka tomu ušetríme pamäťový priestor v dátových štruktúrach.

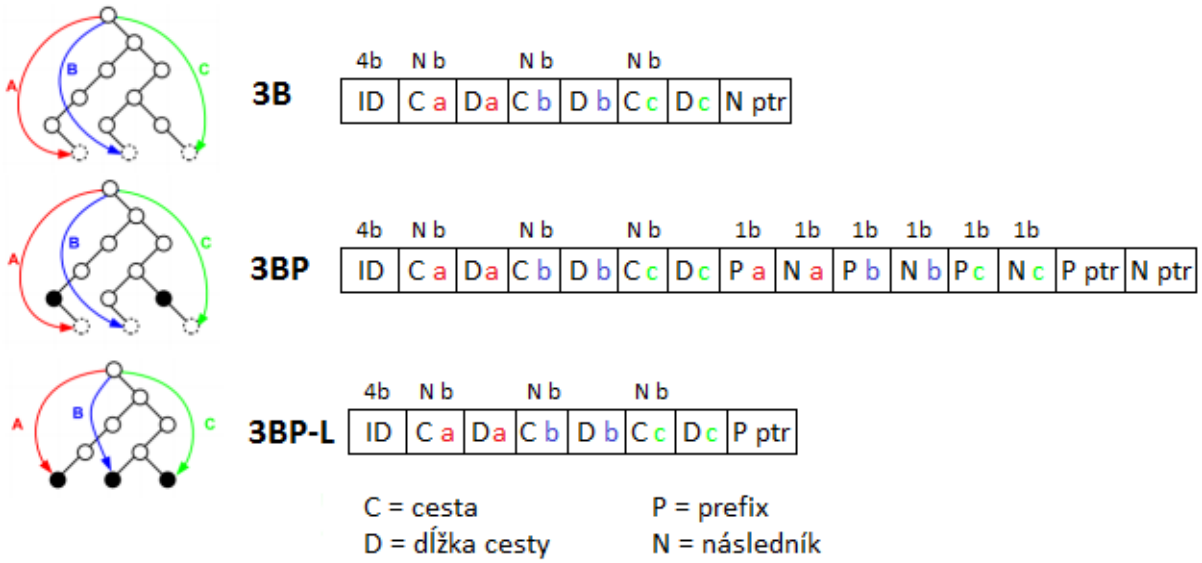
Uvedený popis dátových štruktúr takmer úplne zodpovedá tiež nasledujúcim typom uzlov, ktoré sú znázornené na obrázku 5.3. Jedná sa o uzly, ktoré povoľujú 1 vetvenie, teda obsahujú 2 samostatné vetvy v rámci uzlu. Pri porovnaní dátových štruktúr s predchádzajúcimi (obrázok 5.2) vidíme jednoznačnú podobu. Jediný rozdiel je pridanie jednej vetvy navyše, ktorá si vyžaduje dodatočné 2 dátové položky. Jedná sa o cestu vetvy b a dĺžku tejto cesty. Zvyšné položky, ako tiež logika popisu zostala zachovaná.



Obrázok 5.3: Dátové štruktúry uzlov typu 2B, 2BP a 2BP-L

Veľmi jednoducho by sme potom vedeli odvodiť dátové položky pre posledné typy uzlov, ktoré sú zobrazené na obrázku 5.4. Vidíme, že veľkosť každého uzlu závisí od typu, ale tiež od zvoleného parametra N, ktorý udáva maximálny počet bitov vetvy. Výhodou tohto návrhu je, že je prispôsobený charakteristikám prefixových množín a uzly neplytvajú zbytočne pamäťou. Zároveň je tvar jednotlivých vetiev prakticky ľubovoľný, čo pripomína princíp efektívneho algoritmu SST. Uzly sa teda môžu do istej miery prispôsobovať konkrétnej množine. Pevný počet tvarových typov však bráni prílišnej dynamickosti, ktorá bola hlavných nedostatkom SST. Vďaka tomu je tento návrh

možné jednoducho implementovať v hardwari, keďže vieme každý typ uzlu jednoznačne identifikovať a efektívne ho spracovať.



Obrázok 5.4: Dátové štruktúry uzlov typu 3B, 3BP a 3BP-L

5.1.2 Porovnanie uzlov s TBM

Nová reprezentácia uzlov bola navrhnutá s cieľom eliminovať hlavný nedostatok algoritmu Tree Bitmap – plytvanie pamäťou v riedkych a málo vetvených častiach stromu. Ako bolo dokázané, takéto časti sa vyskytujú veľmi často a to hlavne v protokole IPv6. Ukážme si preto zhodnotenie novej reprezentácie v porovnaní s uzlami Tree Bitmap.

Pre porovnanie použijeme ako referenčnú hodnotu TBM s parametrom n=4 a veľkosť ukazateľov 16 bitov. Veľkosť jedného TBM uzlu potom bude 63 bitov. Takýto uzol pokrýva 4 úrovne stromu Trie, zapuzdruje v sebe maximálne 15 jednobitových uzlov a má maximálne 16 následníkov.

Keďže veľkosť nových uzlov závisí na nastavení parametra N, pokúsme sa nastaviť také hodnoty, aby sme sa čo najviac priblížili veľkosti TBM uzlu – 63 bitov. S ohľadom na predchádzajúci popis dátový štruktúr potom dostávame hodnoty zobrazené v tabuľke 5.1. Všetky údaje sú v bitoch a predstavujú veľkosti jednotlivých položiek dátovej štruktúry uzlu.

V uvedenej tabuľky vidíme, že uzly typu 1B a odvodené majú dĺžku vetvy 20 až 37 bitov. Sú teda veľmi efektívne v nevetvených miestach stromu, kde s pomocou nich dokážeme preskočiť veľké úseky. Pri veľkosti zhodnej s TBM uzlom teda dokážu zostupovať stromom 5-9 krát rýchlejšie. Uzly s dvoma cestami (2B a odvodené) dokážu pokrývať 2 vetvy stromu, kde každá má maximálne 8-16 bitov. Spolu teda takýto uzol dokáže pokryť až 32 unibit uzlov. Oproti TBM je teda zase oveľa efektívnejší, keďže pokrýva 2-násobný počet uzlov a až 4-násobný počet hladín Trie s využitím menšieho množstva pamäte. Uzly, ktoré majú povolené 3 vetvy (3B a odvodené), pokrývajú 12 až 30 unibit uzlov a 4-10 hladín stromu Trie.

	Typ uzlu	Cesta N	$\log_2$ N	Prefix btmp	Child btmp	Prefix ptr	Child ptr	SPOLU [b]
1B	4	37	6	-	-	-	16	63
1BP	4	20	5	1	1	16	16	63
1BPL	4	37	6	-	-	16	-	63
2B	4	2x16	2x4	-	-	-	16	60
2BP	4	2x8	2x3	2	2	16	16	62
2BPL	4	2x16	2x4	-	-	16	-	60
3B	4	3x10	3x4	-	-	-	16	62
3BP	4	3x4	3x2	3	3	16	16	60
3BPL	4	3x10	3x4	-	-	16	-	62

Tabuľka 5.1: Veľkosť jednotlivých dátových položiek navrhnutých uzlov

Vidíme, že naše navrhnuté uzly sú efektívne hlavne v úsekoch s nízkym stupňov vetvenia, čo bol tiež hlavný zámer. Najvýraznejší náskok oproti TBM sme teda dosiahli u uzlov s nevetvenými cestami (1B a odvodené). Uzol typu 3BP je už porovnateľný s uzlom typu TBM. Zavedenie uzlu typu 4BP, ktorý by povoľoval až 4 vetvy v rámci uzlu teda už nemá význam. Jeho efektivita by bola nižšia ako u Tree Bitmap, a preto je výhodnejšie v takýchto úsekoch využiť klasické TBM uzly.

Veľkosť každého uzlu závisí vždy na nastavenom parametri N. V tabuľke boli nastavené tieto parametre tak, aby sme boli schopní jednoducho zhodnotiť nové uzly v porovnaní s TBM. Takéto nastavenie však nemusí byť vždy výhodné. Rozhodujúce je určiť dĺžku vetví jednotlivých uzlov, ktorú považujeme za ideálnu. Od toho potom do veľkej miery závisia celkové pamäťové nároky. U uzlu 1B sú uvedené limity príliš vysoké a pravdepodobne by často boli nevyužité. Preto by bolo výhodnejšie tieto hodnoty znížiť a tým pádom dostaneme tiež nižšiu celkovú veľkosť uzlu. Pri probléme určovania najlepšieho nastavenia parametrov hľadáme kompromis medzi počtom pokrývaných úrovní stromu a celkovou veľkosťou, ktorú bude multi-uzol zaberať.

5.2 Mapovanie

Uvedená reprezentácia prefixovej množiny nám dáva pevnú sadu niekoľkých šablón, s ktorými bude algoritmus ďalej pracovať. Tieto šablóny sa snažíme pri konštrukcii výsledného stromu vhodne mapovať na jednotlivé časti jednobitového stromu Trie. Mapovanie by malo hľadať ideálne rozmiestnenie typov uzlov tak, aby sme dosiahli minimálnu spotrebu pamäte.

Konštrukcia stromu prebieha smerom zhora nadol a začína v koreni Trie. Mapovací algoritmus sa pokúsi v danom mieste namapovať všetky typy uzlov. Každé namapovanie ohodnotí pomocou ceny, ktorá sa počíta nasledovne:

$$price = \begin{cases} \frac{p}{size} & \text{if } \frac{p}{size} > 0 \\ \frac{n}{size} & \text{otherwise} \end{cases}$$

Hodnota *p* je počet prefixových unibit uzlov, ktorý náš uzol pokrýva, *n* je celkový počet unibit uzlov, ktoré náš uzol pokrýva a *size* je veľkosť nášho mapovaného uzlu. Algoritmus potom vyberie taký uzol zo sady, ktorý má najnižšiu cenu. Vyberáme si teda také uzly, ktoré majú najlepší pomer *pokrytie/veľkosť*. Logicky chceme pokryť vždy čo najvyššiu časť stromu, avšak nie za cenu plytania pamäťou. Pseudokód mapovacieho algoritmu je možné popísať nasledovne:

```
1:  Q ← ∅
2:  if root ≠ NULL then
3:      ENQUEUE(Q, root)
4:  while Q ≠ ∅ do
5:      trie ← DEQUEUE(Q)
6:      max_price ← 0
7:      best_type ← NULL
8:      for each type ∈ node_types do
9:          price ← MAP_PRICE(type, trie)
10:         if price > max_price then
11:             max_price ← price
12:             best_type ← type
13:     trie ← MAP(best_type, trie)
14:     for each child ∈ CHILDREN(trie) do
15:         ENQUEUE(Q, child)
```

Mapovanie, ktoré postupuje smerom zhora nadol vždy prehľadáva len malú časť stromu. Nemá informácie o distribúcii a charaktere nižších častí stromu, ktoré by mohol vziať do úvahy. To znamená, že výsledok je len aproximáciou ideálneho namapovania, a preto nedosahuje skutočne minimálnu možnú priestorovú náročnosť. Výhodou tohto riešenia je, že je jednoduché a hlavne dostatočne rýchle. Zložitosť tohto postupu je porovnateľná s vytváraním stromu metódou Tree bitmap. Alternatívou by bolo používať konštrukciu stromu smerom zdola nahor, ktorá by u každého mapovaného uzlu uvažovala komplexné informácie o celom strome. To však vedie na extrémnu výpočtovú zložitosť, ktorá je porovnateľná alebo vyššia ako u algoritmu SST. Takýto postup je možný u malých množín. Pre reálne množiny, ktoré bežne dosahujú stovky tisíc prefixov je ale tento prístup výpočtovo nezvládnuteľný.

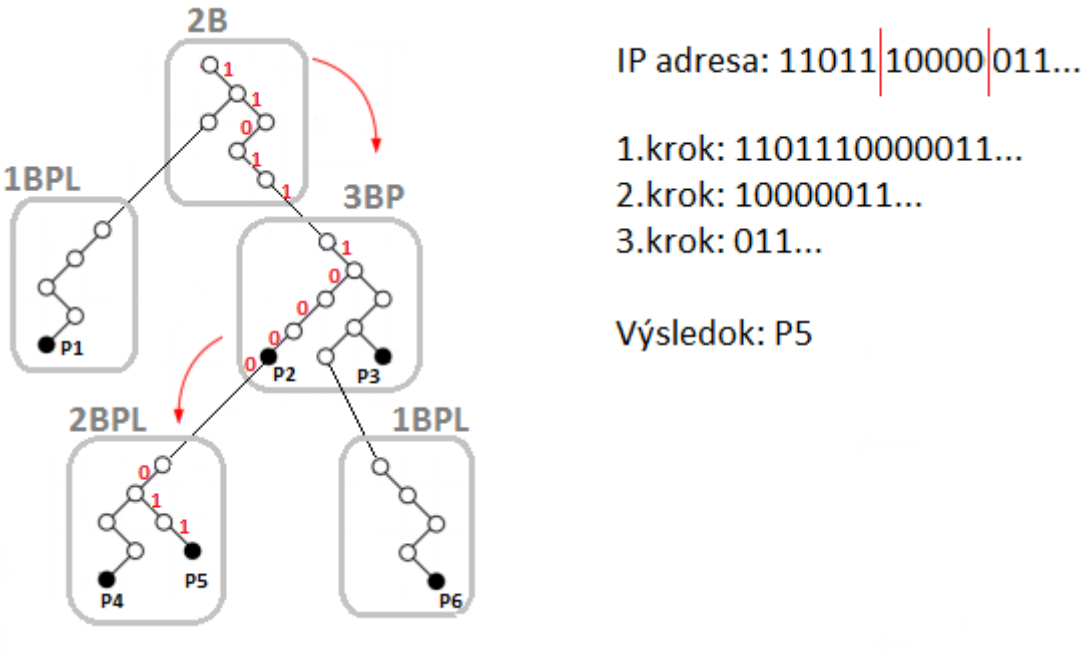
5.3 Spôsob vyhľadávania

Po ukončení konštrukcie stromu je dátová štruktúra pripravená na svoju hlavnú funkciu – vyhľadávanie najdlhšieho zhodného prefixu. Vo svojej podstate sa zase jedná o multibit strom, ktorý spracováva *n* bitov vstupnej IP adresy v každom kroku. Oproti podobným konkurentom však má isté odlišnosti.

Zvyčajný postup vyhľadávania u ostatných algoritmov je aplikovanie jedného opakovaného cyklu až do stanovenej koncovej podmienky. Keďže náš zostavený strom sa skladá z palety rôznych uzlov, nie je možné aplikovať takýto uniformný postup. Vyhľadávanie samozrejme začína v koreni stromu. V každom kroku je prvou činnosťou načítanie ID typu. Na základe toho sa určí, akým

spôsobom sa daný uzol bude spracovávať. Pre každý uzol existuje samostatná funkcia obsluhujúca jeho spracovanie. Po spracovaní sa posunieme do nasledovníka, kde sa postup opakuje až dokým sa nedostaneme na koniec stromu. Tento postup pripomína procesné spracovanie, kde sa opakujú inštrukcie *Fetch -> Decode -> Execute*.

Funkcie obsluhujúce spracovanie jedného uzlu sú v zásade veľmi jednoduché a majú podobný charakter. Binárne cesty uložené v uzle (1-3 cesty podľa typu) porovnávame s aktuálnym úsekom IP adresy. Ak nájdeme zhodu cesty a IP adresy, znamená to, že vyhľadávanie bude pokračovať v synovskom uzle, ktorý nasleduje za danou vetvou. Ak daná vetva obsahuje prefix, aktualizujeme si najdlhší doposiaľ nájdený prefix. Následne je možné odstrániť *n* bitov IP adresy alebo sa posunúť o *n* bitov v adrese, kde *n* je dĺžka cesty danej vetvy. Vyhľadávanie pokračuje v následníkovi a tento postup sa opakuje dokým nenarazíme na vetvu, ktorá nemá žiadneho následníka, alebo v danom uzle nenastane žiadna zhoda cesty a IP adresy. Vyhľadávanie je v takýchto prípadoch ukončené a vrátime výsledok v podobe najdlhšieho nájdeného prefixu. Počet krokov potrebných na uskutočnenie vyhľadania je podobne ako u ostatných stromových algoritmoch závislá na výške stromu. Jednoduchý príklad vyhľadania je ilustrovaný na obrázku 5.5.



Obrázok 5.5: Jednoduchá ukážka vyhľadávania

Rozdiely v spracovaní jednotlivých typov uzlov sú minimálne. Zvyčajne hlavným rozdielom je počet porovnávaných úsekov a informácie o prítomnosti prefixu/následníka. Z charakteru návrhu je často prítomnosť prefixu/následníka v uzle zrejmá už na základe ID typu. Okrem spracovania novo navrhnutých uzlov samozrejme musí byť prítomná ešte funkcia, ktorá dokáže správne dekódovať a spracovať TBM uzol. Informáciu o tom, že sa jedná o TBM uzol opäť získame z ID.

5.4 Aktualizácie stromu

Smerovacia tabuľka nie je statická, ale informácie v nej uložené sa s určitou frekvenciou menia. Tieto zmeny závisia od dynamiky siete, v ktorej sa smerovač nachádza, zvyčajne sa však nevyskytujú príliš často. Zmeny v smerovacej tabuľke sa obvykle dostávajú aj do prefixovej množiny v podobe pridávania, odoberania alebo editácii prefixov. LPM algoritmus by mal byť schopný sa s týmito zmenami vysporiadať a premietnuť ich do svojich dátových štruktúr.
Pri aktualizáciách môžu nastávať rôzne situácie:

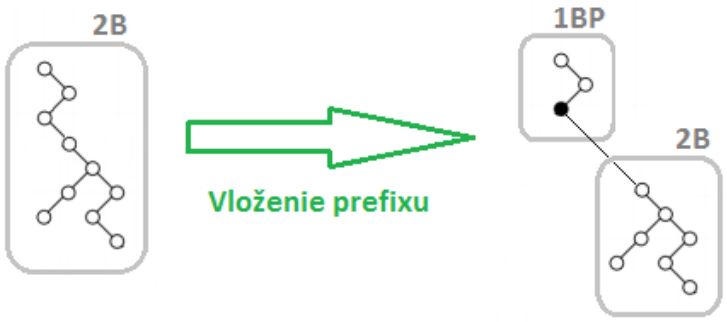
- **Pridávanie nového prefixu** – najnáročnejšia operácia, ktorú si ďalej stručne rozvedieme.
- **Odstránenie existujúceho prefixu** – jednoducho sa zmení bitmapa z 1 na 0, čo značí, že daná vetva už neobsahuje platný prefix. V prípade, že sa jednalo o listový uzol, ktorý neobsahuje žiadne iné prefixy, je možné celý uzol odstrániť. Odstraňovanie môže ďalej pokračovať smerom nahor až dokým nenarazíme na prefixový uzol.
- **Editácia existujúceho prefixu** – jednoduchá operácia, ktorá sa žiadnym spôsobom neprejaví v stromovej konštrukcii. Zvyčajne ide o zmenu next-hop informácie, čiže zmenu ukazateľa.

Potenciálne najnáročnejšou operáciou je operácia pridávania nového prefixu do množiny. V tomto prípade existuje opäť niekoľko možných situácií, ktoré môžu nastať:

- **Pridávanie vyžaduje vytvorenie novej cesty** – uzol, na ktorý sa nová cesta bude napájať sa zmení na uzol s vyšším stupňom vetvenia (1B -> 2B -> 3B -> TBM), prípadne sa listový uzol zmení na nelistový (napr. 2BP-L -> 2BP). Následne sa utvorí celá požadovaná nová cesta. Keďže táto cesta nemá žiadne vetvenia, bude tvorená uzlami typu 1B a zakončená prefixom v uzle typu 1BP.
- **Pridávanie prefixu do TBM uzlu** – algoritmus Tree Bitmap je dobre pripravený na aktualizácie, a preto táto možnosť nevyžaduje žiadne zásahy do stromu. Postup je zhodný s aktualizáciami v algoritme Tree Bitmap.
- **Pridávanie do nových typov uzlov** – ak sa jedná o prefixový uzol a je možné na dané miesto pridať prefix, tak jednoducho zmeníme internú bitmapu z 0 na 1 a pridáme odpovedajúci ukazateľ s next-hop informáciou do poľa prefixov. V opačnom prípade je postup zložitejší a máme 2 možnosti ako túto situáciu riešiť:
    - Uzol, do ktorého pridávame prefix, rozdelíme na 2 menšie nové uzly. Z vrchnej časti sa stane prefixový uzol zakončený novým pridávaným prefixom. Spodná časť je orezaná verzia pôvodného uzlu s rovnakým alebo nižším stupňom vetvenia. Alternatívne je možné celý uzol alebo časť uzlu zmeniť na uzol typu TBM a jednoducho pridať prefix. Situácia pridávania prefixu je zobrazená na obrázku 5.6.
    - V mieste, kde sa pridáva prefix vytvoríme rekonštrukciu celého podstromu pomocou uvedeného mapovacieho algoritmu.

Navrhnutý algoritmus je pripravený na vykonávanie aktualizácií, ktoré vo väčšine prípadov neznamenajú vysoké nároky a prílišný zásah do stromu. Pri vysokej frekvencii pridávania prefixov však efektivita riešenia klesá. V prípade, že sa bude neustále využívať jednoduchá a rýchla verzia pridávania prefixov, kedy sa uzol rozdelí na 2 alebo zmení na TBM, strácame postupom času výhodu maximálnej efektivity využitia pamäte. Zároveň sa zvyšuje výška stromu a teda počet krokov vyhľadávania. Preto je vhodné po vyššom počte aktualizácií vykonať kompletnú rekonštrukciu vyhľadávacieho stromu. Táto operácia je síce výpočtovo a časovo náročnejšia, ale zaručí nám

neustále udržanie maximálnej priepustnosti a minimálnej pamäťovej spotreby. Celková časová náročnosť vykonávania aktualizácií je porovnateľná s algoritmom Tree Bitmap.



Obrázok 5.6: Operácia vloženia prefixu do uzlu typu 2B

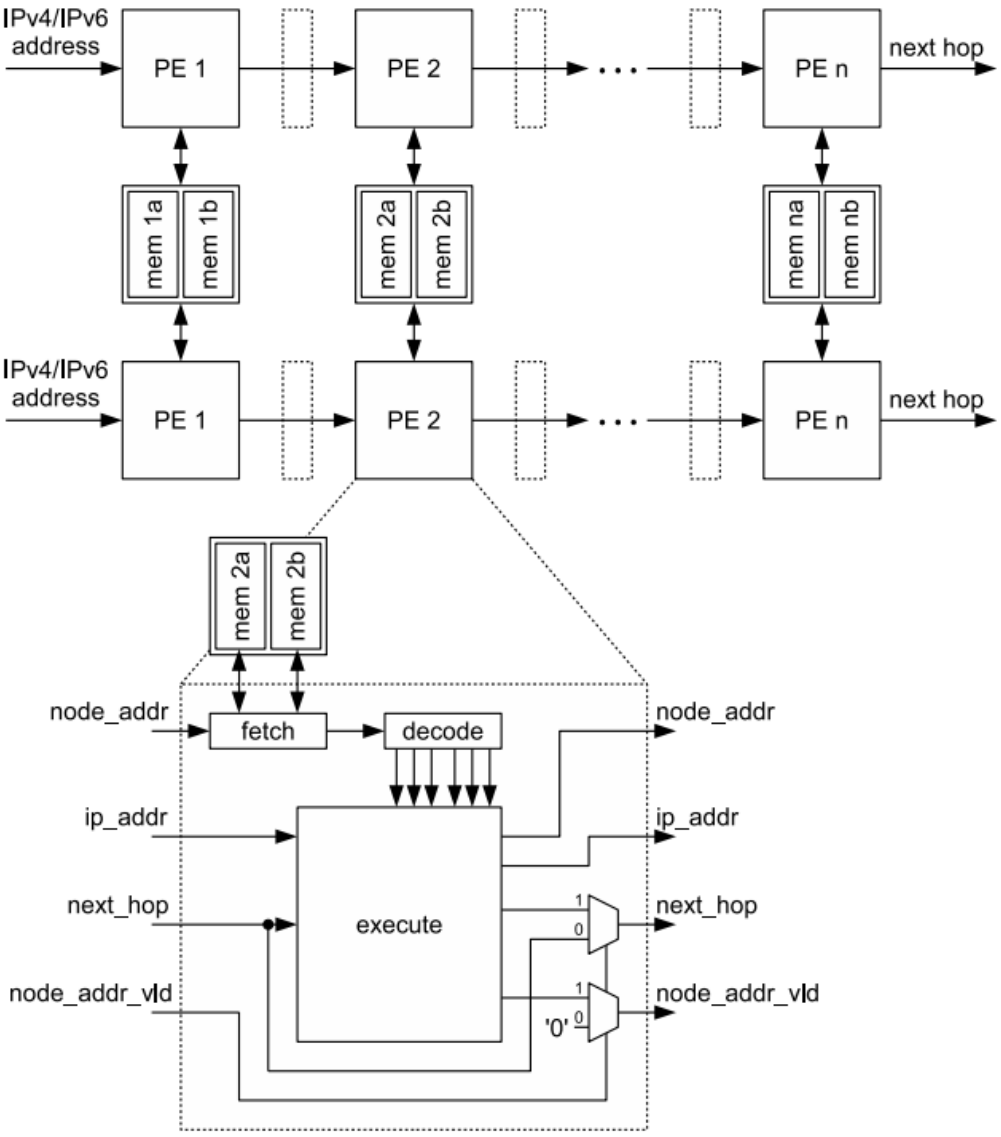
5.5 Návrh HW realizácie

Jednou z priorít pri návrhu algoritmu bolo umožniť jeho hardwarovú implementáciu. Na túto požiadavku bolo nutné myslieť a zohľadňovať ju pri uvažovaných optimalizáciách. V tejto časti stručne predstavím navrhnutú HW architektúru vhodnú pre uvedený algoritmus.

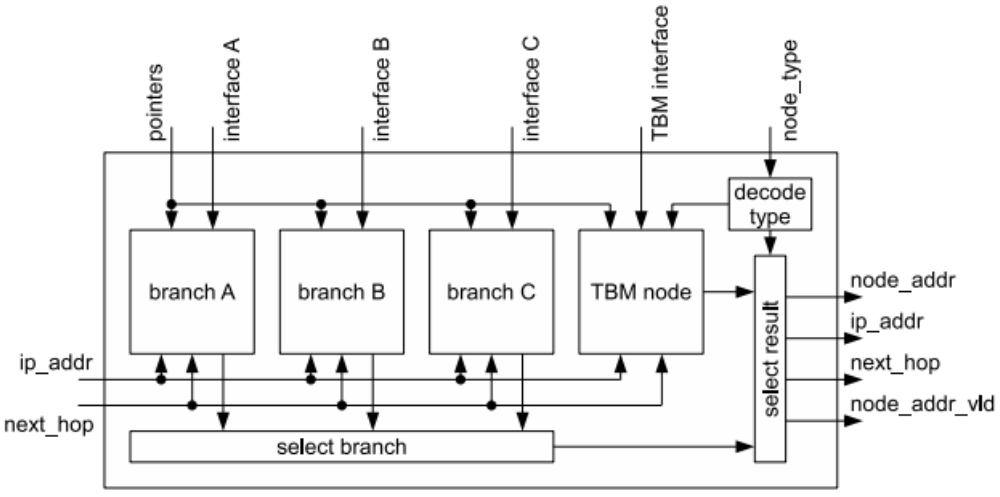
Algoritmus môžeme klasifikovať ako bežnú multibit metódu, kde výsledok dostávame v najhoršom prípade po spracovaní *n* uzlov stromu, kde *n* je výška stromu. Pre dosiahnutie vysokej priepustnosti je nevyhnutné využiť zreťazené spracovanie, kde v každom kroku zreťazenej linky sa vykoná jeden krok vyhľadania. Počet stupňov linky musí byť teda minimálne *n*, čo odpovedá výške vyhľadávacieho stromu. Navrhnutá hardwarová architektúra, ktorá je schopná spracovávať našu prefixovú reprezentáciu je zobrazená na obrázku 5.7.

Táto architektúra pozostáva z dvoch zreťazených liniek a jednej zdieľanej pamäte. Jednotlivé procesné jednotky (PE – processing element) sú uniformné pre všetky stupne pipeline a realizujú jeden krok vyhľadania. Prvý stupeň linky teda predstavuje spracovanie koreňa stromu, ďalší stupeň predstavuje prvú hladinu stromu a podobne. Každý stupeň má priradený blok pamäte, v ktorej sú uložené všetky potrebné dátové štruktúry – v každom stupni sa nachádza 1 hladina vyhľadávacieho stromu. Keďže strom sa v nižších úrovniach stromu rozširuje a obsahuje väčší počet uzlov, predpokladá sa, že nižšie stupne linky potrebujú alokovať väčší pamäťový priestor. Zdieľaná pamäť je dvojportová, takže obe zreťazené linky k nej dokážu pristupovať zároveň, čím dosahujeme dvojnásobný výkon. Pamäťové bloky pozostávajú z dvoch paralelných pamätí, čo umožňuje načítanie každého uzlu v 1 cykle i v prípade, že je uzol rozdelený v 2 dátových slovách.

Na vstupe linky je IP adresa, ktorej prefix budeme vyhľadávať. Postupne prechádza stupňami linky (zostupuje hladinami stromu) až dokým nedosiahne koniec linky, kde dostávame výsledok v podobe najdlhšieho zhodného prefixu alebo priamo odpovedajúcu next-hop informáciu. Po tom, čo adresa opustí prvú procesnú jednotku, je možné začať spracovávať ďalšiu IP adresu. V každom kroku teda dostávame jeden výsledok vyhľadania, ktorý sa však dostaví s určitým oneskorením tvorený stupňami linky.



Obrázok 5.7: Navrhnutá HW architektúra s detailom procesnej jednotky



Obrázok 5.8: Interná štruktúra bloku „execute“ procesnej jednotky

Na obrázku 5.7 je tiež znázornený detail jednej procesnej jednotky. Tá zapuzdruje vyhľadávacie funkcie, ktoré boli spomenuté v kapitole popisujúcej spôsob vyhľadávania. Jedna procesná jednotka je schopná spracovať všetky typy navrhnutých uzlov. V prvom kroku sa vykonáva načítanie uzlu na spracovanie, ktorý sa nachádza na adresa *node_addr*. Následne sa dekóduje typ a uzol sa pripraví na spracovanie. Samotné spracovanie prebehne v bloku *execute*, ktorý je detailnejšie zobrazený na obrázku 5.8. Blok dostáva na vstup IP adresu (*ip_addr*), ktorú paralelne porovnáva so všetkými cestami načítaného uzlu v moduloch *branch A, B, C*. V prípade, že bol načítaný TBM uzol dostávame výsledok z modulu *TBM node*. Všetky operácie prebiehajú paralelne, preto na základe *select* logiky vyberieme platný výsledok. Aktualizuje sa tiež *next_hop* informácia a hodnota *node_addr*, ktorá obsahuje adresu nasledovníka. V prípade, že uzol nemá žiadneho následníka, informuje nás o tom bit na *node_addr_valid*. Celý postup je podobný ako spracovanie inštrukcií v bežnom CPU, kde spracovanie jedného typu uzlu môžeme prirovnať k vykonaniu určitého typu inštrukcie.

Pre potreby HW realizácie musíme návrh predstavený v predchádzajúcich častiach doplniť o niekoľko detailov. Bolo ukázané, že jednotlivé typy uzlov majú variabilnú veľkosť, ktorá závisí na nastavených parametroch N. To so sebou prináša niekoľko prekážok. Pri mapovaní uzlov do pamäte potrebujeme, aby boli uzly zarovnané na určité hodnoty – napr. na mocniny čísla 2. Pri zarovnaní potom vždy dochádza k určitému prebytku nevyužitých bitov. Vhodným nastavením však je možné dosiahnuť minimálne plytvanie v dôsledku zarovnania.

Ďalší problém tiež súvisí s premenlivou veľkosťou uzlov. Pri návrhu sa uvažuje, že každý uzol obsahuje len jeden ukazateľ na potomkov – tak, ako je to u TBM. V algoritme Tree Bitmap však každý uzol má uniformnú veľkosť. Naše uzly sú síce už zarovnané, ale stále má každý typ uzlu odlišnú veľkosť, a preto nie je možné spočítať offset na nasledujúce uzly. Tento problém sa rieši pridaním bimapy veľkostí. Uzly sa zarovnajú na niekoľko pevných veľkostí a táto bitmapa potom jednoznačne určí, aké veľkosti majú nasledovníci uzlu. S týmto postupom dokážeme dopočítať požadovaný offset v poli nasledovníkov. Pri zarovnaní napr. na 8 rôznych veľkostí potom potrebujeme 3 bity na každého následníka. Alternatívnym riešením tohto problému je zarovnať všetky uzly na jednotnú veľkosť – napr. 64 bitov. Táto možnosť nevyžaduje pridávanie ďalších dodatočných dátových položiek, ale môže spôsobovať vyššie plytvanie pamäťou.

6 Optimalizácie kódovania uzlov

V predchádzajúcom popise bolo viditeľné, že návrh algoritmu je značne variabilný a je možné aplikovať množstvo kombinácií vstupných parametrov. V tejto časti sa preto zameriavam na možnosti týchto nastavení a ich dopad na výslednú efektivitu riešenia. Cieľom je vybrať takú množinu uzlov a odpovedajúcich parametrov, aby bola dosiahnutá pokiaľ možno minimálna pamäťová spotreba. Pritom však treba rešpektovať požiadavky HW realizácie.

K dosiahnutiu tohto cieľa bolo nutné vykonať veľké množstvo experimentov. Uvádzam predovšetkým výsledky v kontexte protokolu IPv6. Uskutočnené samozrejme boli ekvivalentné experimenty s množinami protokolu IPv4, aby sme sa nedostali do stavu, kedy by bol algoritmus dobre optimalizovaný pre nový protokol IPv6, ale zaostával v predchádzajúcej verzii protokolu. V prípade záujmu je možné nájsť množstvo dodatočných výsledkov experimentovania na priloženom CD.

Výsledný algoritmus bol implementovaný v jazyku Python a bol začlenený do nástroja Netbench [14]. V využitím tohto nástroja boli vykonané tiež uvedené experimenty.

6.1 Používané prefixové množiny

Pre účely experimentovania a optimalizácií bolo vybraných niekoľko testovacích prefixových množín, ktorých súhrnné vlastnosti zachytáva tabuľka 6.1. Využívali sa najväčšie dostupné tabuľky protokolov IPv4 a IPv6. Reálne množiny boli doplnené syntetickými, ktoré boli získané generovaním spôsobom popísaným v [23]. Výhody tohto prístupu boli stručne predstavené v časti analýzy tejto práce. Vstupom pre generovanie boli uvedené tabuľky protokolu IPv4. Vďaka tomu sme dostali prefixové množiny protokolu IPv6, ktoré dosahujú rozmery porovnateľné s protokolom IPv4. S cieľom zvýšiť diverzitu, boli zaradené množiny z odlišných zdrojov a získané v rôznych dátumoch. Takáto pestrosť by mala dávať relevantné výsledky reprezentujúce mnoho situácií.

	Množina	Prefixy	Zdroj	Dátum
IPv4	rrc00	332 118	ripe [15]	03.06.2010
	IPv4-space	220 779	bgp [1]	21.12.2011
	route-views	442 748	route-views [16]	20.09.2012
IPv6	AS1221	10 518	bgp [1]	21.09.2012
	AS6447	10 814	bgp [1]	21.09.2012
IPv6 gen.	rrc00_ipv6	319 998	generované z rrc00	
	IPv4-space_ipv6	150 157	generované z IPv4-space	
	route-views_ipv6	439 880	generované z route-views	

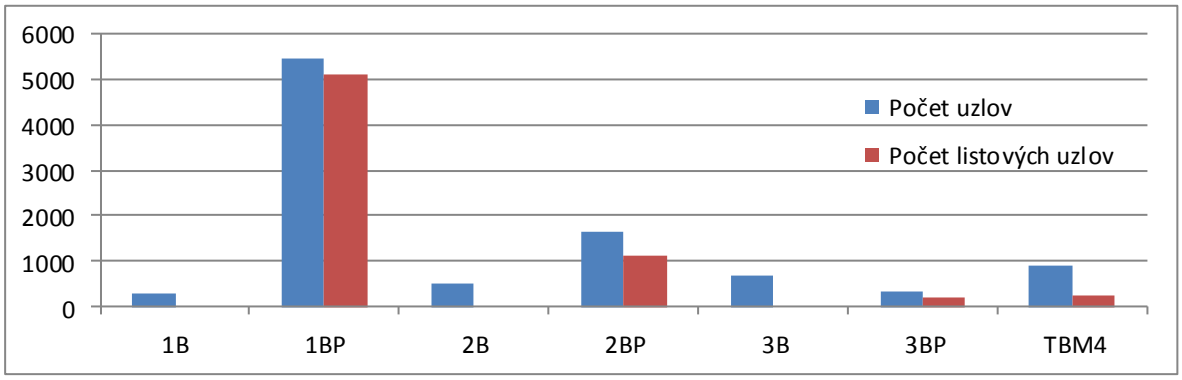
Tabuľka 6.1: Súhrn detailov používaných prefixových množín

6.2 Postup optimalizácie

Táto sekcia ukazuje podstatné experimenty a postup optimalizovania navrhnutých uzlov. Z počiatku uvádzam výsledky z množiny AS1221, ktorá je dostatočne reprezentatívna. Výsledky ostatných množín zdieľajú podobné charakteristické črty. Súhrnné výsledky pre všetky uvedené množiny sa potom nachádzajú v závere tejto podkapitoly. Pri testovaní sa zvyčajne používala logika minimálnych pointerov. To znamená, že na každej úrovni stromu bol použitý minimálny možný ukazateľ, ktorý je potrebný na adresovanie vytvorených uzlov a prefixov. Ak nie je uvedené inak, predpokladá sa teda implicitne tento prístup.

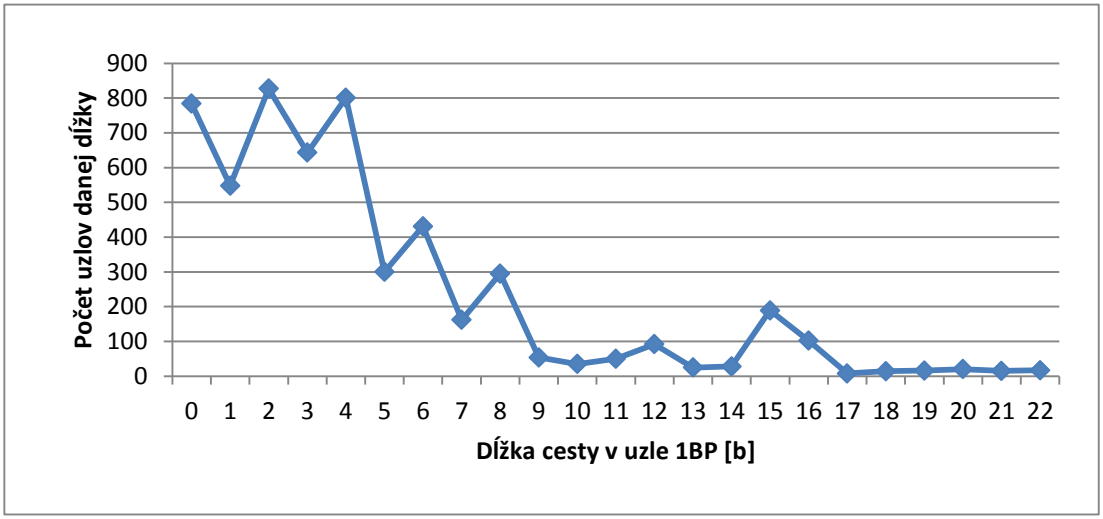
V prvom kroku si ukážeme výsledky algoritmu s využitím základných typov uzlov (1B, 1BP, 2B, 2BP, 3B, 3BP a TBM) – teda bez podpory listových uzlov. Tieto uzly sa uvedenou technikou namapujú na prefixovú množinu a následne získané riešenie môžeme analyzovať. Nastavené hodnoty limitov dĺžok pre jednotlivé uzly sú postupne: 25, 23, 11, 9, 7 a 5 bitov a parameter pre uzol Tree Bitmap bol n=4. Predovšetkým nás zaujíma využitie jednotlivých typov uzlov a celkové pamäťové nároky. Tie predstavujú **62,22 kB** potrebnej pamäte. Túto hodnotu zatiaľ nedokážeme dobre zhodnotiť, preto ju považujeme za referenčnú. Predpokladáme však, že táto situácia nie je ideálna a je možné pomocou optimalizácií uvedenú hodnotu znižovať.

Graf na obrázku 6.1 znázorňuje rozloženie využitia jednotlivých typov uzlov vo vytvorenom strome. Jednoznačne dominantný je uzol typu 1BP, čo potvrdzuje výskyt dlhých nevetvených úsekov. Tento typ uzlu predstavuje najvyšší potenciál pre ďalšie optimalizácie. Červenou farbou sú v grafe zobrazené listové uzly jednotlivých typov. Vidíme, že ich pomerné zastúpenie je veľmi vysoké. V tomto experimente avšak nie sú použité optimalizované listové uzly (1BP-L, 2BP-L, 3BP-L, TBM-L) a ukazuje sa, že ich využitie prinesie pozitívne výsledky.



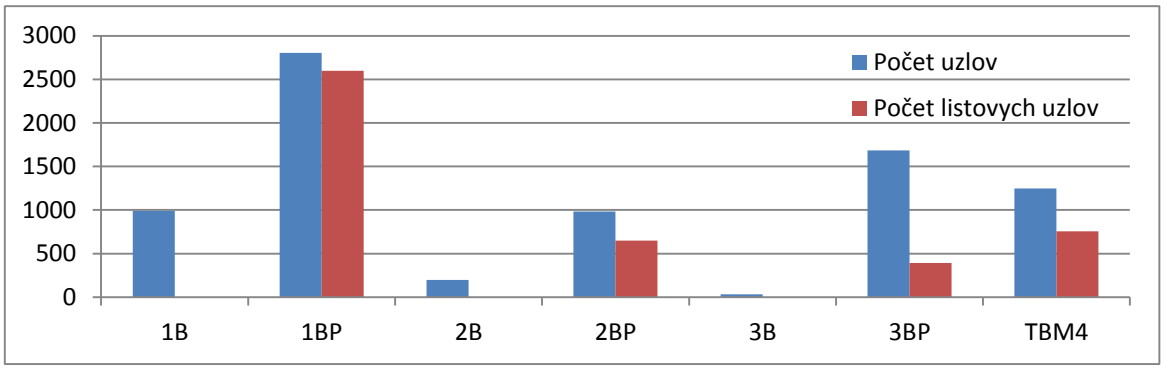
Obrázok 6.1: Využitie základných typov uzlov

Zaujímavou otázkou je vhodnosť nastavených vstupných parametrov. Pri nastavení príliš vysokej hodnoty dochádza k plytvaniu z dôvodu nevyužívania alokovaného priestoru. Pri malých hodnotách zase nevyužívame celý potenciál nových uzlov a navyše sa zvyšuje výška stromu. Graf 6.2 zobrazuje histogram počtu uzlov typu 1BP v závislosti na dĺžke cesty v uzle. V tomto príklade bol limit uzlu 1BP nastavený na 23 bitov, čo sa javí ako zbytočne vysoká hodnota, pretože od hodnoty 9 sa uzly vyskytujú v minimálnom množstve. Prekvapivo vysoký je počet uzlov pre dĺžku 0, ktorá sa nachádza medzi dominujúcimi pozíciami. Dĺžka 0 znamená, že v danom uzle sa nenachádza žiadna cesta, ale len odkaz na platný prefix. Takýto typ uzlu predstavuje tiež potenciál pre optimalizácie.



Obrázok 6.2: Histogram počtu uzlov 1BP v závislosti na dĺžke cesty

Podobným spôsobom by sme dokázali kontrolovať využitie alokovaného priestoru vo všetkých používaných typoch uzlov. Otázka vhodného nastavenia parametrov potom predstavuje optimalizačnú úlohu s cieľom minimalizovať plytvanie v uzloch a teda maximalizovať ich využiteľnosť pri danej prefixovej množine. Nájdenie ideálneho nastavenia parametrov má zásadný vplyv na celkovú efektivitu algoritmu. Obrázok 6.3 ukazuje zhodný experiment ako pri obrázku 6.1 avšak nastavenie parametrov bolo nasledovné: 12, 8, 11, 11, 4 a 12 bitov. Vidíme, že táto jednoduchá zmena spôsobila odlišné zastúpenie jednotlivých uzlov v strome. Celkové pamäťové nároky v tomto prípade klesli na **48,13 kB**, čo je pokles o 22,6% len vďaka zmene argumentov.

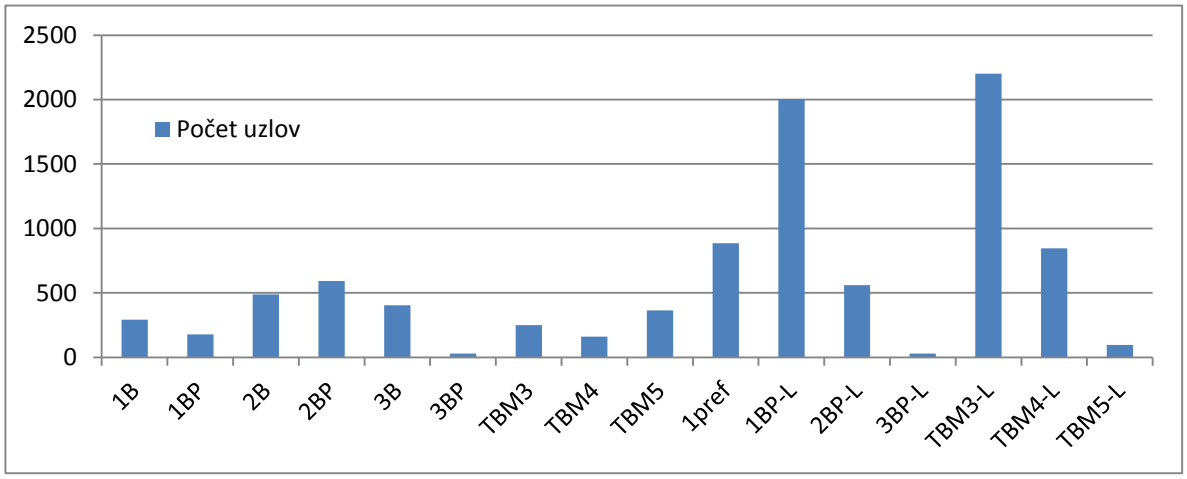


Obrázok 6.3: Využitie základných typov uzlov (odlišné nastavenie parametrov)

Doteraz sme používali vždy Tree Bitmapový uzol s nastavením n=4. Uzol TBM má pokryť situácie s vyšším stupňom vetvenia alebo vyššou hustotou prefixov. Avšak podobne nastavenie vhodného parametra n má aj tu svoj význam. V analýze boli ukázané ako vhodné parametre n=3, 4 alebo 5. V každej množine je výhodnejšie použiť inú hodnotu. Dokonca v jednotlivých častiach stromu je občas výhodné použiť jednu variantu a inokedy druhú. Preto je výhodné povoliť všetky 3 uvedené nastavenia zároveň. Mapovací algoritmus potom na základe ceny sám rozhodne, ktorú variantu TBM uzlu v aktuálnej časti stromu využije.

Z uvedených úvah pripravíme nový experiment, ktorý využíva všetky navrhnuté typy uzlov a vhodné nastavenie vstupných parametrov, ktoré je 25, 23 , 11, 9, 7 a 5 bitov (listové uzly majú limity nastavené zhodne s nelistovou verziou). Oproti uzlom prezentovaným v návrhu algoritmu sa tu navyše objavuje uzol typu „1pref“. Ten predstavuje možnosť uzlu 1BP s nulovou dĺžkou cesty, ako bolo uvedené vyššie. Zastúpenie jednotlivých typov uzlov vidíme na obrázku 6.4.

Najvýraznejšiu pozíciu získal uzol typu 1BP-L spolu s TBM3-L. Vidíme teda, že zavedenie listových uzlov malo význam a pozitívny dopad. Vyšší počet uzlov zvyšuje rozmanitosť a pravdepodobnosť, že mapovací algoritmus vytvorí optimálnu konštrukciu stromu. Pamäťové nároky pre túto variantu predstavujú **42,97 kB**, čo je pokles o ďalších 10,7%.



Obrázok 6.4: Rozloženie využitia všetkých typov uzlov

Súhrnné výsledky pre všetky používané prefixové množiny sú k dispozícii v tabuľke 6.2. Prvý stĺpec obsahuje výsledky pre základné typy uzlov so zhodným nastavením parametrov ako bolo použité vyššie. Druhý stĺpec potom analogicky všetky typy dostupných uzlov. Pamäťové nároky sú uvedené v kB. Výsledky v tabuľke potvrdzujú, že zvýšená variabilita v podobe väčšieho množstva typov uzlov prináša veľmi pozitívne výsledky. Oproti používaniu len základných typov uzlov tak dokážeme zvýšiť efektivitu a znížiť pamäťové nároky v niektorých prípadoch až o takmer 70%.

Množina	Základné uzly	Všetky uzly	Ušetrená pamäť
rrc00	1 598,37	667,40	**58,25%**
IPv4-space	996,84	313,15	**68,59%**
route-views	2 102,78	863,62	**58,93%**
AS1221	62,22	42,97	**30,93%**
AS6447	64,02	44,15	**31,04%**
rrc00_ipv6	3091,76	2 321,52	**24,91%**
IPv4-space_ipv6	1 434,47	1 086,23	**24,28%**
route-views_ipv6	4 244,52	3 203,13	**24,53%**

Tabuľka 6.2: Súhrnné výsledky uvedených experimentov

6.3 Zarovnané uzly

Doteraz sme sa pri optimalizovaní uzlov pohybovali do istej miery v teoretickej rovine a aplikovali sme myšlienkové postupy s cieľom minimalizovať pamäťové nároky algoritmu. Neustále je potrebné myslieť tiež na budúcu hardwarovú implementáciu a zohľadňovať jej požiadavky. Ako bolo popísané v návrhu, HW realizácia si kladie určité dodatočné podmienky, predovšetkým na zarovnanie uzlov.

Experimenty v predchádzajúcej sekcii nebrali do úvahy zarovnanie uzlov a výsledkom je, že výsledná veľkosť uzlov je veľmi rozmanitá. Každý uzol mohol mat obecne ľubovoľnú veľkosť. Prvým a najjednoduchším spôsobom ako túto situáciu riešiť je zarovnať všetky uzly na jednotnú veľkosť. Tento prístup je pomerne priamočiary, avšak nie je príliš efektívny. Pri jeho aplikovaní musíme nastaviť také hodnoty parametrov, aby bol počet nevyužitých bitov v každom uzle pokiaľ možno minimálny. Pri zarovnávaní uzlov však vždy bude dochádzať k istému plytvaniu.

Druhou predstavenou možnosťou bolo pridanie ďalšej bitmapy do každého uzlu a zarovnávanie na niekoľko možných veľkostí. Tento prístup už zvyšuje variabilitu, čo je predpokladom na nižšie plytvanie a vyššiu efektivitu, avšak za cenu vyššej réžie pri spracovávaní uzlov. V novej bitmape sú uložené informácie o všetkých následníkoch daného uzlu. To predstavuje problém pre uzly typu TBM4 a TBM5, ktoré obsahujú veľký počet následníkov. Uzol TBM4 by potreboval dodatočných 32 bitov a TBM5 až 64 bitov pri zarovnaní na 4 typy veľkostí (resp. 48 a 96 bitov pri zarovnaní na 8 typov). Z toho dôvodu výrazne strácajú svoju efektivitu a rozhodol som sa tieto 2 typy z experimentov vynechať. Ich listové varianty TBM4-L a TBM5-L však nepotrebujú žiadnu dodatočnú bitmapu, preto tieto sú v experimentoch zachované. Prestávame používať tiež uzol typu „1pref“. Tento uzol bojoval hlavne svojou malou veľkosťou (19 bitov). Avšak po aplikovaní zarovnania narastie jeho veľkosť na 32 bitov, čím dostávame veľkosť zhodnú s TBM3 a tento typ uzlu stráca svoj pôvodný zmysel.

Ďalšiu komplikáciu spôsobuje použitie minimálnych ukazateľov. Keďže potrebná veľkosť ukazateľa je dopredu neznáma, nie je známa ani veľkosť uzlu. Bolo by teoreticky možné aplikovať mapovanie a následne uzly zarovnávať na najbližšiu povolenú veľkosť. Súčasne by bolo treba aktualizovať bitmapy všetkých predchodcov. To predstavuje výrazné zvýšenie nárokov celého algoritmu. Navyše výsledky vykonaných experimentov ukázali, že použitie minimálneho ukazateľa (pri aplikovaní zarovnania) zníži nároky len o približne 1%. Preto v nasledujúcich experimentoch uvažujem použitie fixnej veľkosti 23 bitových pointrov, ktoré sú dostatočné vo všetkých testovaných prípadoch.

Po aplikovaní uvedených zmien sme schopní vykonávať experimenty, ktoré zohľadňujú všetky požiadavky HW realizácie. Simuláciou takto nastavených parametrov dostávame výsledky, ktoré už nie sú len teoretické, ale zodpovedajú skutočnosti a je možné ich dosiahnuť na navrhnutej architektúre. Hľadanie vhodných nastavení parametrov je však teraz omnoho náročnejšia úloha, keďže je potrebné zohľadňovať tiež zarovnanie a minimalizáciu nevyužitých bitov. Navrhol som 2 druhy zarovnaní (s krokom 8 a s krokom 16 bitov). Hodnoty parametrov pre obe varianty sa nachádzajú v tabuľke 6.3 a sú uvedené v bitoch.

	Zarovnané po 8 bitoch			Zarovnané po 16 bitoch		
Typ	**Limit cesty**	**Nezarovnaná veľkosť**	**Zarovnaná veľkosť**	**Limit cesty**	**Nezarovnaná veľkosť**	**Zarovnaná veľkosť**
1B	24	56	56	17	48	48
1BP	19	72	72	13	64	64
1BPL	20	48	48	20	48	48
2B	16	72	72	14	64	64
2BP	10	80	80	11	80	80
2BPL	12	55	56	15	61	64
3B	11	78	80	12	78	80
3BP	5	80	80	6	80	80
3BPL	7	53	56	9	62	64
TBM3	3	75	80	3	67	80
TBM3L	3	30	32	3	30	48
TBM4L	4	38	40	4	38	48
TBM5L	5	54	56	5	54	64

Tabuľka 6.3: Nastavenie parametrov pri aplikovaní zarovnania

V tabuľke 6.4 sú následne zhrnuté dosiahnuté výsledky po aplikovaní daných parametrov a zostavení prefixového stromu pomocou navrhnutého algoritmu. Zahrnuté sú tiež výsledky pri použití jednoduchého zarovnania všetkých uzlov na jednotnú veľkosť (64b). Vidíme, že najlepšie výsledky dosahujeme pri variante zarovnávania uzlov po 8 bitoch. Je to spôsobené opäť vyššou variabilitou a nižším plytvaním, ktoré sa objavuje vplyvom zarovnávania. Pri zarovnaní po 16 bitoch alebo dokonca na jednotnú veľkosť je počet nevyužitých bitov podstatne vyšší a efektivita klesá. Pri porovnaní získaných hodnôt s predchádzajúcimi (teoretickými) experimentmi je viditeľný nárast pamäťovej náročnosti, ktorý činí v priemere 25%. Tento nárast je logický a je spôsobený povahou nutných zmien potrebných pre HW realizáciu.

Množina	**Zarovnané po 8b**	**Zarovnané po 16b**	**Všetky uzly 64b**
rrc00	**791,35**	910,95	1 065,00
IPv4-space	**446,42**	537,17	595,27
route-views	**972,48**	1 129,95	1 319,63
AS1221	**59,47**	61,12	78,06
AS6447	**61,73**	63,33	80,52
rrc00_ipv6	**2 658,04**	2 671,65	3 390,15
IPv4-space_ipv6	**1 301,52**	1 302,68	1 599,16
route-views_ipv6	**3 629,94**	3 650,93	4 638,78

Tabuľka 6.4: Pamäťové nároky algoritmu po aplikovaní zarovnania [kB]

7 Zhodnotenie výsledkov

V predchádzajúcej kapitole sme vykonali optimalizácie zakódovania prefixového stromu. Cieľom bolo pokúsiť sa nájsť optimálne nastavenie vstupných parametrov navrhnutého algoritmu. Boli ukázané viaceré varianty spolu s ich výsledkami, ktoré boli podložené experimentmi.

V tejto časti práce sa zameriavam na celkové zhodnotenie výsledného návrhu. Na začiatku sa nachádza porovnanie s algoritmami Tree Bitmap a Shape Shifting Tree, kde je uvádzaná celková pamäťová spotreba pri konkrétnych prefixových množinách. Pri porovnávaní používam parametre algoritmu, ktoré v predchádzajúcej časti vykazovali najlepšie hodnoty, avšak s rešpektovaním HW požiadaviek. Jedná sa teda o uzly s využitím bitmapy veľkostí, ktoré sú zarovnané po 8 bitoch.

Do zhodnotenia som sa rozhodol zaradiť tiež porovnanie s algoritmom Prefix Partitioning (DPP), ktorý patrí medzi state-of-the-art v oblasti LPM. Je to teda jeden z najefektívnejších dnes známych LPM algoritmov. Pre možnosť porovnať pamäťovú efektivitu nášho návrhu s týmto algoritmom je potrebné zaviesť odlišnú metodiku vyhodnocovania. Tá spočíva v určení pomeru Q, ktorý značí počet Bytov pamäti potrebných na uloženie 1 Bytu prefixu. Hodnotu tohto pomeru teda získame nasledovne:

$$Q = \frac{celkové\ pamäťové\ nároky\ algoritmu\ [B]}{pamäť\ potrebná\ pre\ uloženie\ prefixov\ v\ množine\ [B]}$$

S využitím tejto metriky som uskutočnil porovnanie navrhnutého algoritmu s uvedenými 3 metódami, čím získame celkový prehľad o pamäťovej efektivite prístupu. V závere kapitoly sú napokon diskutované výkonnostné parametre riešenia na odpovedajúcej HW architektúre.

7.1 Porovnanie s TBM

Pri porovnávaní s algoritmom Tree Bitmap som vychádzal z výsledkov analýzy (obrázok 4.7), ktorá ukázala, že najlepšia hodnota kroku je n=5. Pri tejto hodnote sme dosahovali najnižšie hodnoty algoritmu TBM a jednotlivé uzly nemali prílišnú veľkosť. Veľkosť každého uzlu je pri tomto parametri rovná 63 bitov + 2 ukazatele. Uzly TBM majú teda vyššiu veľkosť ako uzly predstaveného algoritmu, ktoré majú maximálne 32 až 80b vrátane ukazateľov. Pri algoritme TBM som povolil pri každej množine používať minimálny ukazateľ, ktorým je možné adresovať vytvorené uzly a prefixy. Beriem teda v úvahu najnižšie hodnoty, aké je možné algoritmom TBM dosiahnuť. Napriek tomu výrazne zaostáva za naším riešením.

Výsledky porovnania sú prehľadne zhrnuté v tabuľke 7.1. Oproti výsledkom TBM dosahujeme pamäťovú úsporu približne 34 až 78%. Táto úspora je vyššia pri protokole IPv6, kde dokáže nová reprezentácia uzlov využiť svoj potenciál v plnej miere. Predstavený algoritmus má oproti metóde TBM ďalšie výhody, ktoré boli diskutované v predchádzajúcich častiach. Predovšetkým je to pripravenosť a vhodnosť nasadenia v protokole IPv6. Hlavnou motiváciou a cieľom však bola minimalizácia pamäťových požiadaviek, čo bolo úspešne dosiahnuté.

Množina	TBM (n=5) [kB]	Navrhnutý algoritmus [kB]	Ušetrená pamäť
rrc00	1 211,18	791,35	**34,66%**
IPv4-space	712,76	446,42	**37,37%**
route-views	1 492,76	972,48	**34,85%**
AS1221	159,43	59,47	**62,70%**
AS6447	165,82	61,73	**62,77%**
rrc00_ipv6	11 627,44	2 658,04	**77,14%**
IPv4-space_ipv6	6 069,01	1 301,52	**78,55%**
route-views_ipv6	15 967,74	3 629,94	**77,27%**

Tabuľka 7.1: Pamäťové nároky navrhnutého algoritmu v porovnaní s algoritmom TBM

7.2 Porovnanie s SST

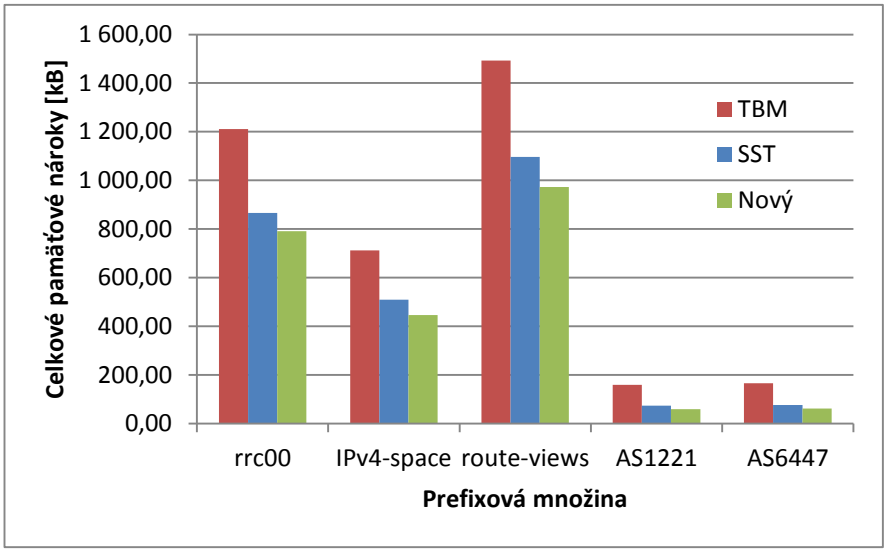
Pri nastavení parametrov metódy Shape Shifting Trie používam hodnotu analogickú metóde TBM, teda K=32. Pri tomto nastavení dokáže jeden uzol SST pokryť maximálne $2^5 = 32$ uzlov Trie. Ako ukázala analýza, táto hodnota dosahuje veľmi dobré výsledky a zároveň drží veľkosť jedného uzlu ešte v akceptovateľnom limite. Veľkosť SST uzlu pri tomto nastavení dosahuje 129b + 2 ukazatele. Uzly algoritmu SST teda majú oproti navrhnutému riešeniu niekoľkonásobne vyššiu veľkosť. Zvyšovanie hodnoty K by viedlo na ďalšie narastenie veľkosti uzlov, čím sa zvyšuje samozrejme náročnosť spracovania a réžia. Problematické by mohlo byť tiež mapovanie týchto uzlov do pamäte.

Pri algoritme SST sme povolili použitie minimálnych ukazateľov, kým v predstavenom prístupe pracujeme s fixnou veľkosťou 23b pre každý odkaz. Ako je vidieť z tabuľky 7.2, navrhnutý algoritmus napriek tomuto znevýhodneniu prekonáva svojou pamäťovou úsporou i metódu SST, ktorá bola navrhnutá práve s ohľadom na minimalizáciu pamäťových nárokov. Dosiahnuté výsledky sú viditeľné tiež v grafickej podobe na obrázku 7.1, na ktorom sú pre názornosť zahrnuté tiež výsledky metódy Tree Bitmap. Nenachádzajú sa tu však výsledky pre generované množiny IPv6 a to z dôvodu extrémnej výpočtovej náročnosti metódy SST. Tá je akceptovateľná pre menšie množiny, avšak s narastajúcim počtom uzlov Trie sa neúmerne zvyšuje. Získanie výsledkov pre uvedené 3 generované množiny by trvalo rádovo niekoľko mesiacov.

Množina	SST (K=32) [kB]	Navrhnutý algoritmus [kB]	Ušetrená pamäť
rrc00	866,31	791,35	**8,65%**
IPv4-space	510,13	446,42	**12,49%**
route-views	1096,87	972,48	**11,34%**
AS1221	73,56	59,47	**19,16%**
AS6447	77,14	61,73	**19,98%**

Tabuľka 7.2: Pamäťové nároky navrhnutého algoritmu v porovnaní s algoritmom SST

Novo predstavený algoritmus dokázal svojou úspornosťou prekonať metódu Shape Shifting Trie a to vo všetkých testovaných prefixových množinách. Tento fakt považujem za významný úspech, keďže algoritmus SST bol sám o sebe navrhnutý ako veľmi úsporný. Navrhnutý algoritmus znížil celkové potrebné pamäťové nároky o približne 8 až 20%. Navyše výsledky SST sa pohybujú v teoretickej rovine, keďže neexistuje odpovedajúca HW architektúra. Nová reprezentácia teda jednoznačne zvíťazila v priestorovej efektivite. Avšak na rozdiel od SST drží svoju dynamickosť a variabilitu v rozumnej miere. Vďaka tomu si navrhnutá metóda dokázala udržať charakter, ktorý umožňuje metódu implementovať v podobe predstavenej hardwarovej architektúry.



Obrázok 7.1: Porovnanie pamäťových nárokov návrhu, TBM a SST

7.3 Pamäťová efektivita

V tejto časti sa nachádza zhodnotenie pamäťovej efektivity s využitím metriky uvedenej v úvode kapitoly. Jedná sa o pomer efektívnosti Q, ktorý udáva počet Bytov pamäti potrebných na uloženie jedného Bytu prefixu. Hlavným cieľom bolo porovnať efektivitu navrhnutého algoritmus s metódou Prefix Partitioning (DPP), ktorá patrí medzi state-of-the-art v oblasti LPM. Pre názornosť a objektívne zhodnotenie boli zahrnuté tiež algoritmy TBM a SST.

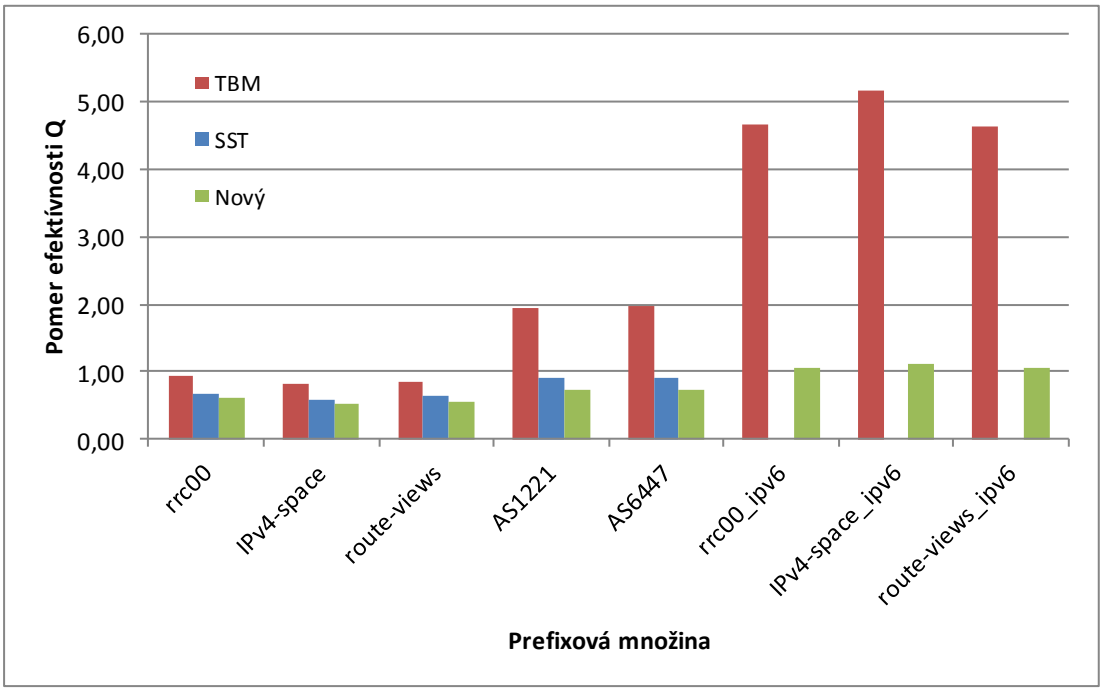
Hodnota parametru Q pre navrhnutý algoritmus je zobrazená v tabuľke 7.3 a na grafe obrázku 7.2. Metóda DDP uvádza [9] priemernú hodnotu tohto pomeru 1,01 zhodnú pre množiny protokolu IPv4 i IPv6. Z našich získaných výsledkov vidíme, že pri reálnych sadách predstavený algoritmus dosahuje výrazne lepšie hodnoty tohto ukazateľa, ktoré sa pohybujú na úrovni 0,52 až 0,73. Pri generovaných množinách protokolu IPv6 sú výsledky na porovnateľnej úrovni. Prezentované výsledky teda jasne potvrdzujú, že navrhnutý algoritmus víťazí svojou pamäťovou efektivitou i nad algoritmom DPP.

Z výsledkov tiež vidieť, že pri algoritme TBM pomer efektívnosti Q rastie pri použití protokolu IPv6 a najhoršie výsledky dosahuje pri generovaných množinách IPv6. To opäť potvrdzuje

nevhodnosť nasadenia algoritmu TBM s novým IP protokolom. Pritom pri novej navrhnutej metóde nemá zásadný vplyv veľkosť prefixovej množiny alebo verzia použitého protokolu. Vo všetkých prípadoch si zachováva výborné hodnoty, ktorými prekonáva ostatné algoritmy.

Množina	Navrhnutý algoritmus	TBM (n=5)	SST (K=32)
rrc00	**0,61**	0,93	0,67
IPv4-space	**0,52**	0,83	0,59
route-views	**0,56**	0,86	0,63
AS1221	**0,72**	1,94	0,90
AS6447	**0,73**	1,96	0,91
rrc00_ipv6	**1,06**	4,65	-
IPv4-space_ipv6	**1,11**	5,17	-
route-views_ipv6	**1,06**	4,65	-

Tabuľka 7.3: Pomer Q - počet bytov pamäti/počet bytov prefixov



Obrázok 7.2: Pomer efektívnosti Q pre nový navrhnutý algoritmus, TBM a SST

7.4 Výsledky HW syntézy

Doteraz sme sa zameriavali predovšetkým na priestorovú zložitosť návrhu a minimalizáciu pamäťových nárokov. Hlavnou motiváciou pritom bolo dosiahnuť takú úroveň, aby nebolo nutné využívať externú pamäť na ukladanie dátových štruktúr. Veľmi podstatným výkonovým faktorom je priepustnosť, ktorú sme schopní pri nasadení dosiahnuť. Tú však nie je možné odvodiť z výsledkov jednoduchej simulácie, ale záleží na konkrétnej HW architektúre.

Z toho dôvodu bola uvedená hardwarová architektúra implementovaná a syntetizovaná pre cieľové zariadenie Xilinx Virtex-6 XC6VSX475T FPGA. Výsledky syntézy v podobe využitých zdrojov a odhadovanej pracovnej frekvencie sa nachádzajú v tabuľke 7.4. Napriek vysokému počtu stupňov zreťazenej linky (23) je možné celú uvedenú architektúru vložiť do cieľového FPGA. Obe linky sú schopné vykonať jedno vyhľadanie v každom hodinovom cykle. Pri uvedenej odhadovanej frekvencii teda dostávame približne 230 miliónov vyhľadaní za sekundu. Takže navrhnuté riešenie disponuje priepustnosťou až 140 Gbps.

Na základe znalosti frekvencie a počtu stupňov pipeline vieme spočítať tiež celkovú latenciu linky. Každý procesný element PE obsahuje v sebe 5 dodatočných stupňov, z čoho dostávame 5*23=115. Jedna zreťazená linka teda pozostáva celkovo zo 115 stupňov. Keďže spracovanie jedného stupňa trvá 8,66ns, dostávame celkovú latenciu riešenia, ktorá je 995,9ns. Tento čas uplynie od počiatku spracovávania paketu až do získania výsledku v podobe next-hop adresy. Po túto dobu musí byť paket uložený v bufferi, v ktorom čaká na výsledok. Veľkosť tohto bufferu musí byť minimálne 14,42 kB.

Časť architektúry	LUT		Registre		Frekvencia [MHz]
	Počet	% zo všetkých	Počet	% zo všetkých	
1PE	4 038	1,36 %	1 827	0,31 %	115,41
1x pipeline (23 PE)	92 874	31,21 %	42 021	7,06 %	115,41
2x pipeline (46 PE)	185 748	62,42%	84 042	14,12 %	115,41

Tabuľka 7.4: Využitie zdrojov a odhad pracovnej frekvencie navrhnutej architektúry

Tieto výkonnostné parametre sa ukazujú byť veľmi priaznivé. Daný návrh predstavuje univerzálne riešenie, ktoré disponuje nízkou pamäťovou náročnosťou a vysokou priepustnosťou až 140 Gbps. Navrhnutá architektúra využíva jednoduché princípy príbuzné klasickým procesným jednotkám. Nevyžaduje zložité operácie v podobe hashovacích funkcií (na ktorých sú založené niektoré prístupy), ktoré komplikujú návrh a réžiu spracovania. Ďalšou pozitívnou vlastnosťou je tiež nízka energetický náročnosť vďaka využívaniu výhradne interných zdrojov.

8 Záver

Cieľom tejto práce bolo navrhnúť nové algoritmické riešenie operácie vyhľadávania najdlhšieho zhodného prefixu (LPM). Hlavnými požiadavkami návrhu bola nízka pamäťová spotreba, vysoká rýchlosť vyhľadávania a priepustnosť v kontexte protokolu IPv6. Všetky vytýčené ciele sa mi podarilo splniť a navyše som pre vytvorený algoritmus navrhol odpovedajúcu hardwarovú architektúru. Vytvorené riešenie vykazuje výborné vlastnosti, vďaka ktorým môže nájsť dobré uplatnenie v smerovačoch vysokorýchlostných sietí.

Úvodným a podstatným krokom ešte pred samotnou tvorbou tejto technickej správy bolo získanie dostatočného prehľadu a odborných informácií. Dostupné zdroje som preštudoval a spracoval v podobe teoretického rozboru, ktorý tvorí vstupný bod do problematiky. Snažil som sa poskytnúť všetky znalosti, ktoré sú nevyhnutné, ale zároveň ich podať v dostatočne prehľadnej a výstižnej forme. Po zasadení problematiky do kontextu a oboznámení sa s princípom LPM, som sa zameral na popis vybraných algoritmov. Tie sú postavené na stromovej štruktúre, ktorá skrýva veľký potenciál pri vyhľadávaní a zvyčajne poskytuje dobrú vyváženosť všetkých výkonnostných kritérií, ktorými sú hlavne rýchlosť vyhľadania a pamäťová náročnosť. Vysvetlil som princípy fungovania jednotlivých algoritmov spolu so stručným zhodnotením ich kladných a záporných vlastností.

Ďalším krokom bola podrobná analýza existujúcich algoritmov z pohľadu ich efektivity so zreteľom na nasadenie v kontexte protokolu IPv6. Pri analýze som sledoval teoretické výkonnostné hodnoty v podobe horných odhadov zložitosti, ale i výsledky algoritmov nad reálnymi prefixovými sadami získaných zo smerovačov rozsiahlych autonómnych sietí. Vybral som dva najperspektívnejšie algoritmy z pohľadu pamäťovej zložitosti, ktoré ukazujú potenciálny priestor pre návrh nového algoritmu. Sú to metódy Tree Bitmap a Shape Shifting Trie. Pri návrhu nového algoritmu som vychádzal z kladných vlastností uvedených algoritmov a snažil som sa odstrániť ich nedostatky. Významná bola tiež analýza reálnych prefixových tabuliek. Z nej som získal cenné informácie v podobe identifikovania charakteristických rysov prefixových množín protokolu IPv6. Viackrát bolo potvrdené, že typicky v prefixovom strome prevládajú dlhé nevetvené úseky, prípadne časti s malým stupňom vetvenia. Táto skupina spolu s listovými uzlami tvoria hlavný priestor pre aplikovanie optimalizácií.

Na základe vykonanej analýzy som navrhol a implementoval nový algoritmus tak, aby umožnil vysokú priepustnosť, nízku spotrebu pamäte a jednoduchú hardwarovú implementáciu. Motiváciou a hlavnou myšlienkou pritom bolo minimalizovať predovšetkým pamäťovú náročnosť, aby bolo možné všetky dátové štruktúry uložiť v internej pamäti FPGA. Nový algoritmus, ktorý som navrhol, spočíva v odlišnej reprezentácii prefixovej množiny. Základom je sada niekoľkých typov uzlov, ktoré sú optimalizované podľa výsledkov analýzy. Navrhnuté uzly sa mapujú na prefixový strom s cieľom dosiahnuť optimálne pokrytie – teda s cieľom minimalizovať počet potrebných uzlov a výšku stromu. Pre tento algoritmus som tiež navrhol odpovedajúcu HW architektúru, ktorá využíva zreťazené spracovanie a princíp jej práce pripomína procesorové spracovávanie inštrukcií.

Veľmi podstatným krokom bolo dôkladné otestovanie a zhodnotenie vytvoreného návrhu. Algoritmus bol implementovaný v jazyku Python a zaradený do nástroja Netbench, v rámci ktorého prebiehali tiež experimenty. Využíval som rôzne druhy prefixových tabuliek protokolov IPv4 a IPv6 pre simuláciu reálnych podmienok. Dosiahnuté výsledky boli konfrontované s ďalšími algoritmami. Nový navrhnutý algoritmus je výrazne úspornejší ako Tree Bitmap (až o 78%), ktorý je dnes bežne nasadený v CISCO smerovačoch. Predstavený algoritmus prekonáva dokonca i metódu Shape

Shifting Trie (až o 20%), ktorá bola navrhovaná s cieľom minimalizovania pamäťovej spotreby. V porovnaní s metódou Prefix Partitioning predstavený algoritmus opäť jednoznačne víťazí pri reálnych množinách. V prípade generovaných množín IPv6 dosahujeme porovnateľné výsledky. Podľa dostupných informácií teda vytvorený návrh prekonáva svojou pamäťovou efektivitou všetky dnes známe prístupy. Taktiež predstavená HW architektúra, ktorá bola implementovaná pre technológiu Virtex-6 FPGA, vyniká výbornými vlastnosťami. Využíva jednoduché princípy založené predovšetkým na zreťazenom spracovaní a dosahuje priepustnosť až 140Gbps.

Navrhnutý algoritmus vykazuje výborné vlastnosti a javí sa ako veľmi perspektívny. Vďaka tomu vzbudil záujem tiež na odborných IEEE konferenciách. Uvedený návrh bol prezentovaný na medzinárodnej konferencii IEEE DDECS [11]. Návrh bol tiež podaný na konferencii FPL, kde v danej chvíli čaká na výsledky recenzie. Možnosti pokračovania tejto práce sú široké. Je neustále možné hľadať optimalizácie návrhu a naďalej sa snažiť znižovať pamäťové nároky. Ďalší priestor predstavuje mapovací algoritmus – nájsť teoretické optimálne pokrytie a navrhnúť zmeny, aby sme sa tomuto optimu priblížili. Najväčší potenciál však predstavuje HW architektúra. Tu je možné pokračovať cestou znižovania potrebných zdrojov a zvyšovania vyhľadávacieho výkonu. Zaujímavou myšlienkou je tiež využiť dynamickú rekonfiguráciu FPGA, vďaka čomu by sme dokázali spravodlivo rozdeliť alokovanú pamäť pre jednotlivé stupne linky a tiež reagovať na zmeny v sieti.

Literatúra

[1] BGP Routing Table Analysis Reports [online]. Dostupné na URL: <http://bgp.potaroo.net/>

[2] Eatherton, W. N. *Hardware-based Internet Protocol Prefix Lookups*. Saint Louis, Missouri: Washington University, 1999. s. 28-29. Diplomová práca.

[3] Eatherton, W., Varghese, G. a Dittia, Z. *Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates*. ACM SIGCOMM Computer Communication Review. Apríl 2004, roč. 34, č. 2. s. 97-122. ISSN 0146-4833.

[4] Degermark, M. aj. *Small forwarding tables for fast routing lookups*. ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication. September 1997, ISBN 0-89791-905-X, s. 3-14.

[5] Fuller, V., aj. *Classless Inter-Domain Routing (CIDR): Address Assignment and Aggregation Strategy* [RFC 1519]. September 1993. Dostupné na URL: <http://www.ietf.org/rfc/rfc1519.txt>

[6] Halsall, F. *Computer networking and the internet*. Edinburg: Addison-Wesley, 2005, 803 s. ISBN 0-321-26358-8.

[7] Hinden, R. M., Deering, S. E. *Internet Protocol Version 6 (IPv6) Addressing Architecture* [RFC 3513]. Apríl 2003. Dostupné na URL: <http://www.ietf.org/rfc/rfc3513.txt>

[8] Chao, H a Bin L. *High performance switches and routers*. Hoboken: John Wiley & Sons, 2007, 613 s. ISBN 978-0-470-05367-6.

[9] Le, H., Prasanna, V. *Scalable Tree-based Architectures for IPv4/v6 Lookup Using Prefix Partitioning*. IEEE Transactions on Computers, Júl 2012, roč. 61, č.7, s, 1026-1039. ISSN 0018-9340.

[10] Loshin, P. *IPv6 theory, protocol, and practice*. 2nd edition. San Francisco: Morgan Kaufmann, 2004, 536 s. ISBN 1-55860-810-9.

[11] Matoušek, J., Skačan, M., Kořenek, J. *Towards Hardware Architecture for Memory Efficient IPv4/IPv6 Lookup in 100 Gbps Networks*. IEEE Design and Diagnostics of Electronic Circuits and Systems DDECS'2013, Karlovy Vary, CZ, 2013.

[12] Nilsson, S., Karlsson, G. *IP-Address Lookup Using LC-Tries*. IEEE Journal on Selected Areas in Communications. Jún 1999, roč. 17, č.6, s. 1083-1092. ISSN 0733-8716.

[13] Postel, J. *Internet Protocol* [RFC 791]. September 1981. Dostupné na URL: <http://www.ietf.org/rfc/rfc791.txt>

[14] Puš, V., Tobola, J, Košař, V, Kaštil, J., Kořenek, J. *Netbench: Framework for Evaluation of Packet Processing Algorithms*. Seventh ACM/IEEE Symposium on Architecture for Networking and Communi-cations Systems (ANCS'11). IEEE Computer Society, Október 2011, s. 95-96, ISBN 978-0-7695-4521-9.

[15] RIS Raw Data – Prefixset rrc00 [online]. Dostupné na URL: <http://data.ris.ripe.net/rrc00/>

[16] Route Views Archive Project [online]. Dostupné na URL: <http://archive.routeviews.org/>

[17] Sanchez, M., Biersack, W. a Dabbous, W. *Survey and taxonomy of IP address lookup algorithms*. IEEE Network. Marec 2001, roč. 15, č.2, s. 8-23.

[18] SixXS – IPv6 Deployment & Tunnel Broker [online]. Dostupné na URL: <http://www.sixxs.net>

[19] Song, H., Turner, J. a Lockwood, J. *Shape Shifting Tries for Faster IP Route Lookup*. In Proceedings of the 13TH IEEE International Conference on Network Protocols. Washington, USA: IEEE Computer Society, 2005. s. 358-367. ISBN 0-7695-2437-0.

[20] Sportack, M. A. *Směrování v sítích IP*. Vyd. 1. Brno: Computer Press, 2004, 351 s. ISBN 80-251-0127-4.

[21] Srinivasan, V., Varghese, G. *Faster IP lookups using controlled prefix expansion*. ACM SIGMETRICS Performance Evaluation Review. Jún 1998, roč. 26, č.1, s. 1-10.

[22] Waldvogel, M., Varghese, G., Turner, J. a Plattner B. *Scalable High Speed IP Routing Lookups*. ACM SIGCOMM Computer Communication Review. Október 1997, roč. 27, č.4, s. 25-37, ISSN 0146-4833.

[23] Wang, M., Deering, S., Hain, T., Dunn, L. *Non-random Generator for IPv6 Tables.* HOTI '04 Proceedings of the High Performance Interconnects. August 2004, s. 35-40, ISBN 0-7803-8686-8.